UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
09/582,297	04/08/2002	Kenneth Carbone	06975-029006	1661

7590 02/01/2006

Fish & Richardson
601 Thirteenth Street N W
Washington, DC 20005

EXAMINER	
OSMAN, RAMY M	
ART UNIT	PAPER NUMBER
2157	

DATE MAILED: 02/01/2006

Please find below and/or attached an Office communication concerning this application or proceeding.

Office Action Summary	Application No.	Applicant(s)	
	09/582,297	CARBONE ET AL.	
	Examiner	Art Unit	
	Ramy M. Osman	2157	

-- The MAILING DATE of this communication appears on the cover sheet with the correspondence address --

Period for Reply

A SHORTENED STATUTORY PERIOD FOR REPLY IS SET TO EXPIRE 3 MONTH(S) OR THIRTY (30) DAYS, WHICHEVER IS LONGER, FROM THE MAILING DATE OF THIS COMMUNICATION.

- Extensions of time may be available under the provisions of 37 CFR 1.136(a). In no event, however, may a reply be timely filed after SIX (6) MONTHS from the mailing date of this communication.
- If NO period for reply is specified above, the maximum statutory period will apply and will expire SIX (6) MONTHS from the mailing date of this communication.
- Failure to reply within the set or extended period for reply will, by statute, cause the application to become ABANDONED (35 U.S.C. § 133). Any reply received by the Office later than three months after the mailing date of this communication, even if timely filed, may reduce any earned patent term adjustment. See 37 CFR 1.704(b).

Status

1)☒ Responsive to communication(s) filed on *08 April 2002*.

2a)☐ This action is **FINAL**. 2b)☒ This action is non-final.

3)☐ Since this application is in condition for allowance except for formal matters, prosecution as to the merits is closed in accordance with the practice under *Ex parte Quayle*, 1935 C.D. 11, 453 O.G. 213.

Disposition of Claims

4)☒ Claim(s) *1-69* is/are pending in the application.

4a) Of the above claim(s) _____ is/are withdrawn from consideration.

5)☐ Claim(s) _____ is/are allowed.

6)☒ Claim(s) *1-69* is/are rejected.

7)☐ Claim(s) _____ is/are objected to.

8)☐ Claim(s) _____ are subject to restriction and/or election requirement.

Application Papers

9)☐ The specification is objected to by the Examiner.

10)☒ The drawing(s) filed on *08 April 2002* is/are: a)☒ accepted or b)☐ objected to by the Examiner.

Applicant may not request that any objection to the drawing(s) be held in abeyance. See 37 CFR 1.85(a).

Replacement drawing sheet(s) including the correction is required if the drawing(s) is objected to. See 37 CFR 1.121(d).

11)☐ The oath or declaration is objected to by the Examiner. Note the attached Office Action or form PTO-152.

Priority under 35 U.S.C. § 119

12)☐ Acknowledgment is made of a claim for foreign priority under 35 U.S.C. § 119(a)-(d) or (f).

a)☐ All b)☐ Some * c)☐ None of:

1.☐ Certified copies of the priority documents have been received.

2.☐ Certified copies of the priority documents have been received in Application No. _____.

3.☐ Copies of the certified copies of the priority documents have been received in this National Stage application from the International Bureau (PCT Rule 17.2(a)).

* See the attached detailed Office action for a list of the certified copies not received.

Attachment(s)

1) ☒ Notice of References Cited (PTO-892)
2) ☐ Notice of Draftsperson's Patent Drawing Review (PTO-948)
3) ☒ Information Disclosure Statement(s) (PTO-1449 or PTO/SB/08) Paper No(s)/Mail Date 10/30/03 6/23/00
4) ☐ Interview Summary (PTO-413) Paper No(s)/Mail Date. _____.
5) ☐ Notice of Informal Patent Application (PTO-152)
6) ☐ Other: _____.

DETAILED ACTION

Status of Claims

1. This communication is responsive to application filed on April 8, 2002. Claims 1-69 are pending.

Claim Rejections - 35 USC § 102

2. The following is a quotation of the appropriate paragraphs of 35 U.S.C. 102 that form the basis for the rejections under this section made in this Office action:

> A person shall be entitled to a patent unless –
>
> (b) the invention was patented or described in a printed publication in this or a foreign country or in public use or on sale in this country, more than one year prior to the date of application for patent in the United States.

3. Claims 1-6,8-13,16-35,37-42,44-48,51-68 rejected under 35 U.S.C. 102(b) as being anticipated by Stevens (TCP/IP Illustrated, Volume 1: The Protocols, 1994).

4. In reference to claims 1,37, Stevens teaches a method and computer readable medium for asynchronously transferring a plurality of data objects between client and host devices, the method comprising:

transmitting to a client device a plurality of identifiers for data objects, each identifier corresponding to a different one of the data objects to be transferred (pgs 224-226);

transferring over a network between the host and client devices a data frame that includes an identifier and at least a portion of the corresponding data object; and repeating the data frame transfers until the plurality of data objects have been transferred (pgs 12,224-226 & 228, Stevens discloses sending IP datagrams, where each datagram includes a sequence identifier

corresponding to a TCP segment (i.e. data object); the datagrams are sent until all segments are transmitted).

5. In reference to claims 2,38, Stevens teaches the method and computer readable medium of claims 1,37 respectively, wherein at least two sequential transfers of a data frame include transferring frames with different identifiers (pgs 224-226).

6. In reference to claims 3,39, Stevens teaches the method and computer readable medium of claims 1,37 respectively, wherein the transfers of the portions of at least two data objects are interleaved (pgs 224-226,275-288).

7. In reference to claims 4,40, Stevens teaches the method and computer readable medium of claims 1,37 respectively, further comprising: transmitting a data transfer request from the client device to the host device, the transmission of a plurality of identifiers being in response to the data transfer request (pg 231; bottom half of the page).

8. In reference to claims 5,41, Stevens teaches the method and computer readable medium of claims 1,37 respectively, wherein the transfers are downloads (pgs 229,231,275-288; downloads are inherent feature of data transfer).

9. In reference to claims 6,42, Stevens teaches the method and computer readable medium of claims 1,37 respectively, wherein a portion of the transfers are uploads and a portion of the transfers are downloads, the uploads and downloads being interleaved (pgs 229,231,239,275-288; uploads and downloads are inherent features of data transfers).

10. In reference to claims 8,43, Stevens teaches the method and computer readable medium of claims 1,37 respectively, further comprising: transmitting to the client device a size for data

frames before the transfers, the data frames transferred being of said size (pgs 236-238, Stevens discloses MSS).

11. In reference to claims 9,44, Stevens teaches the method and computer readable medium of claims 1,37 respectively, further comprising: transmitting a frame count to the client device, the frame count corresponding to the number of data frames that the client device can transfer without receiving a request for more data frames (pgs 236-238,275-288).

12. In reference to claims 10,45, Stevens teaches a method and computer readable medium for asynchronously transferring a plurality of data objects between client and host devices, the method comprising:

transmitting to a client device a plurality of identifiers and routings of one or more handling processes, each identifier corresponding to one of the data objects (pgs 1-3,6,12, Stevens discloses handling processes as the applications specified in the Application Layer of the TCP/IP protocol suite);

transferring between the client and host devices a first data frame that includes a first identifier, a routing of a first handling process, and at least a portion of the data object corresponding to the first identifier; transferring between' the client and host devices a second data frame that includes a second identifier, a routing of a second handling process, and at least a portion of the data object corresponding to the second identifier; and repeating the data frame transfers until the plurality of data objects have been transferred (pgs 12,224-226 & 228, Stevens discloses sending IP datagrams, where each datagram includes a sequence identifier corresponding to a TCP segment (i.e. data object); the datagrams are sent until all segments are transmitted).

13. In reference to claims 11,46, Stevens teaches the method and computer readable medium of claims 10,45 respectively, further comprising: writing the portions of the data objects to first and second storage locations to which the respective first and second identifiers are assigned (pg 224, it is inherent that the data segments will be written into storage).

14. In reference to claims 12,47, Stevens teaches the method and computer readable medium of claims 11,46 respectively, wherein the writes of the first and second portions of the data objects corresponding to the first and second identifiers are controlled by the first and second handling processes, respectively (pg 224).

15. In reference to claims 13,48, Stevens teaches the method and computer readable medium of claims 10,45 respectively, wherein the first and second handling processes handle uploads of data objects for first and second data objects (pgs 1-3,6,12).

16. In reference to claim 16, Stevens teaches the method of claim 10, wherein the request for more data frames includes the routing of the first handling process (pgs 231-232).

17. In reference to claims 17,51, Stevens teaches a method and computer readable medium for asynchronously transferring data between host and client devices, comprising:

receiving from a client device a frame requesting a data transfer session (pgs 12,231-232);

sending to the client device a frame defining a session protocol that assigns an identifier to each data object (pgs 12,236-238); and

transferring a plurality of data frames between the client and host devices, each data frame comprising a data portion of a data object and an identifier assigned to the data object including said data portion (pgs 12,224-226,275-288).

18. In reference to claims 18,52, Stevens teaches the method and computer readable medium of claims 17,51 respectively, wherein the transferring of data frames includes a data upload (pgs 229,231).

19. In reference to claims 19,53, Stevens teaches the method and computer readable medium of claims 18,52 respectively, further comprising: writing a particular data portion to a storage volume assigned to a particular identifier in response to receiving a data frame including the particular identifier and data portion, unique data objects being assigned to each storage volume (pg 224, it is inherent that the data segments will be written into storage).

20. In reference to claims 20,54, Stevens teaches the method and computer readable medium of claims 17,51 respectively, further comprising: receiving a second frame from the client device requesting a second data transfer session (pgs 231-232); sending a second frame to the client device defining a second session protocol that assigns an identifier to each data object of the second session (pgs 236-238); transferring a plurality of second data frames between the client and host devices, each second data frame including a second data portion and an identifier assigned to a data object including the second data portion (pgs 224-226).

21. In reference to claims 21,55, Stevens teaches the method and computer readable medium of claims 20,54 respectively, wherein the transfers of first and second data frames are interleaved (pgs 224-226,232,239,275-288).

22. In reference to claims 22,56, Stevens teaches the method and computer readable medium of claims 20,54 respectively, wherein the transfers of second data frames are downloads from the host device (pgs 224-226,232).

22. In reference to claims 23,57, Stevens teaches the method and computer readable medium of claims 17,51 respectively, further comprising: receiving a frame from a second client device requesting a second data transfer session(pgs 231-232); sending a frame to the second client device defining a second session protocol that assigns an identifier to each second data object of the second session (pgs 236-238); and transferring a plurality of second data frames between the second client and host devices, each second data frame including a second data portion of a second data object and an associated identifier (pgs 224-226).

23. In reference to claims 24,58, Stevens teaches the method and computer readable medium of claims 17,51 respectively, further comprising: sending to the client device a routing for a handling program assigned to each data object; and wherein each data frame includes the routing of the handling program assigned to the data object therein (pgs 1-3,6).

24. In reference to claims 25,59, Stevens teaches the method and computer readable medium of claims 24,58 respectively, wherein first and second data objects are assigned first and second handling programs, respectively (pgs 1-3,6,12).

25. In reference to claims 26,60, Stevens teaches the method and computer readable medium of claims 24,58 respectively, further comprising: writing a particular data portion to a storage volume assigned to a particular identifier in response to receiving a data frame including the particular identifier and data portion. unique data objects being assigned to each storage volume (pg 224).

26. In reference to claim 27, Stevens teaches the method of claim 26, further comprising: controlling the write with the handling program assigned to the data object being written (pgs 1-3,6).

27. In reference to claims 28,61, Stevens teaches a method and computer readable medium for transmitting data over a network between host and client devices, the method comprising:

receiving from a client device a frame requesting one of a data upload session and a data download session (pgs 12,229,231);

establishing a session protocol in response to receiving the frame from the client device (pgs 12,229-232);

transmitting to the client device a frame defining the session protocol (pgs 12,229-232);

receiving from the client device a data frame conforming to the protocol if the frame from the client device requested an upload; and transmitting to the client device a data frame conforming to the protocol if the frame from the client device requested a download (pgs 12,224-226,229-232).

28. In reference to claims 29,62, Stevens teaches the method and computer readable medium of claims 28,61 respectively, wherein the establishing a session protocol includes: assigning a handling program and a storage location to each data object identified in the frame requesting a session; and wherein the transmitting to the client device a frame defining the session protocol includes sending an identifier for the storage location and a routing for the handling program assigned to each data object (pgs 12,224-226,229,231).

29. In reference to claims 30,63, Stevens teaches the method and computer readable medium of claims 28,61 respectively, wherein the transmission of a frame defining the session protocol includes: transmitting a size for data frames to the client device (pgs 236-238).

30. In reference to claims 31,64, Stevens teaches the method and computer readable medium of claims 28,61 respectively, wherein the transmission of a frame defining a session protocol

includes transmitting a frame count, the frame count being the number of data frames that the client can send prior to receiving a request for more data (pgs 275-288).

31. In reference to claims 32,65, Stevens teaches the method and computer readable medium of claims 28,61 respectively, wherein the transmission of a frame defining a session protocol includes transmitting a format for a command to abort to the client device; and further comprising terminating the session in response to receiving the command to abort from the client device (pgs 231-234).

32. In reference to claims 33,66, Stevens teaches the method and computer readable medium of claims 28,61 respectively, wherein the transmission of a data frame comprises: receiving a frame including an identifier for a storage location, a routing of a handling program, and data to store in the identified storage location; wherein the transmission of a session protocol includes transmitting to the client device the identifier and the routing of the handling program assigned to each data object of the session (pgs 1-3,6,12).

33. In reference to claims 34,67, Stevens teaches the method and computer readable medium of claims 28,61 respectively, wherein the act of transmitting a data frame further comprises: receiving a second message including a second identifier for a second storage location and data to store in the second storage location; and wherein the transmitting a frame defining the session protocol includes transmitting the second identifier to the client device (pgs 224-226).

34. In reference to claims 35,68, Stevens teaches the method and computer readable medium of claims 28,61 respectively, where in the act of establishing includes assigning a storage location and associated identifier to each data object identified in the frame requesting a session (pgs 224-226, 229-234).

Claim Rejections - 35 USC § 103

35. The following is a quotation of 35 U.S.C. 103(a) which forms the basis for all obviousness rejections set forth in this Office action:

> (a) A patent may not be obtained though the invention is not identically disclosed or described as set forth in section 102 of this title, if the differences between the subject matter sought to be patented and the prior art are such that the subject matter as a whole would have been obvious at the time the invention was made to a person having ordinary skill in the art to which said subject matter pertains. Patentability shall not be negatived by the manner in which the invention was made.

36. Claims 7,14,15,36,43,49,50,69 rejected under 35 U.S.C. 103(a) as being unpatentable over Stevens (TCP/IP Illustrated, Volume 1: The Protocols, 1994) in view of Applicant Admitted Prior Art (AAPA, disclosure to application 09/582,297).

37. In reference to claims 7,43, Stevens teaches the method and computer readable medium of claims 1,37 respectively, wherein the transfers of data frames stop at a preselected frame count in the absence of a request for more data frames from a device that receives the data frames.

"Official Notice" is taken that requesting files is old and well-known in the art. It is well known that when a client requests a file that only that particular file is sent, whereupon another file is not sent unless a client requests another file (Stevens, pgs 12 & 229; and AAPA disclosure pgs 1-2). It would have been obvious for one of ordinary skill in the art to stop transfers in the absence of a request because that is a standard of practice in IP communication.

38. In reference to claims 14,49, Stevens teaches the method and computer readable medium of claims 13,48 respectively, wherein the first and second data objects include data for first and second images, respectively ("Official Notice" is taken that image files are well-known in the art. It is well known that files can contain any form of digital data which includes images, among

other things). It would have been obvious for one of ordinary skill in the art to make the data objects include image file data because that is one of the standard types of data that can be digitized and transferred via IP communication.

39. In reference to claims 15,50, Stevens teaches the method and computer readable medium of claims 10,45 respectively, wherein the transfers of data frames including the first identifier stop at a preselected frame count in the absence of a request for more data frames from a device that receives the data frames

"Official Notice" is taken that requesting files is old and well-known in the art. It is well known that when a client requests a file that only that particular file is sent, whereupon another file is not sent unless a client requests another file (Stevens, pgs 12 & 229; and AAPA disclosure pgs 1-2). It would have been obvious for one of ordinary skill in the art to stop transfers in the absence of a request because that is a standard of practice in IP communication.

40. In reference to claims 36,69, Stevens teaches the method and computer readable medium of claims 35,68 respectively, wherein the data objects comprise: a first image file; and a second image file ("Official Notice" is taken that image files are well-known in the art. It is well known that files can contain any form of digital data which includes images, among other things). It would have been obvious for one of ordinary skill in the art to make the data objects include image file data because that is one of the standard types of data that can be digitized and transferred via IP communication.

Any inquiry concerning this communication or earlier communications from the examiner should be directed to Ramy M. Osman whose telephone number is (571) 272-4008. The examiner can normally be reached on M-F 9-5.

If attempts to reach the examiner by telephone are unsuccessful, the examiner's supervisor, Ario Etienne can be reached on (571) 272-4001. The fax phone number for the organization where this application or proceeding is assigned is 571-273-8300.

Information regarding the status of an application may be obtained from the Patent Application Information Retrieval (PAIR) system. Status information for published applications may be obtained from either Private PAIR or Public PAIR. Status information for unpublished applications is available through Private PAIR only. For more information about the PAIR system, see http://pair-direct.uspto.gov. Should you have questions on access to the Private PAIR system, contact the Electronic Business Center (EBC) at 866-217-9197 (toll-free).

RMO
December 10, 2005

ABDULLAHI SALAD
PRIMARY EXAMINER

Substitute Form PTO-1449 (Modified) U.S. Department of Commerce Patent and Trademark Office	Attorney's Docket No. 06975-029006	Application No. 09/582,297
Information Disclosure Statement by Applicant (Use several sheets if necessary) (37 CFR §1.98(b))	Applicant Kenneth Carbone et al.	
	Filing Date April 8, 2002	Group Art Unit 2631

U.S. Patent Documents

Examiner Initial	Desig. ID	Document Number	Publication Date	Patentee	Class	Subclass	Filing Date If Appropriate
	AA	5,481,710	01/02/1996	Keane et al.			
	AB	5,717,860	02/10/1998	Graber et al.			
	AC	6,208,995	03/27/2001	Himmel et al.			
	AD	6,310,630	10/30/2001	Kulkarni et al.			
	AE	6,544,295	04/08/2003	Bodnar			

Foreign Patent Documents or Published Foreign Patent Applications

Examiner Initial	Desig. ID	Document Number	Publication Date	Country or Patent Office	Class	Subclass	Translation Yes	Translation No
	AF	EP0643541A2	03/15/1995	EPO				
	AG	EP0645688A1	03/29/1995	EPO				
	AH	EP0749081A1	12/18/1996	EPO				
	AI	WO97/25804	07/17/1997	PCT				

Other Documents (include Author, Title, Date, and Place of Publication)

Examiner Initial	Desig. ID	Document
	AJ	M. Bieber et al., "Fourth generation hypermedia: some missing links for the World Wide Web", International Journal of Human-Computer Studies, July 1997, Academic Press, UK, vol. 47, no. 1, pages 31-65, XP002101194.
	AK	L. Tauscher et al., "How people revisit web pages: empirical findings and implications for the design of history systems", International Journal of Human-Computer Studies, July 1997, Academic Press, UK, vol. 47, no. 1, pages 97-137, XP002101195.
	AL	M. Bieber, "Providing information systems with full hypermedia functionality", Proceeding of the Twenty-Sixth Hawaii International Conference on System Sciences (CAT. NO. 93TH0501-7), Wailea, HI, USA, 5-8, January 1993, pages 390-400, vol. 3, XP002101196.
	AM	"Routing of Incoming Calls in An X.25 System," IBM Technical Disclosure Bulletin, vol. 32, No. 11, April 1990, pages 370-372.

Examiner Signature	Date Considered 12/10/05

EXAMINER: Initials citation considered. Draw line through citation if not in conformance and not considered. Include copy of this form with next communication to applicant.

534 Rec'd PCT/PTC 23 JUN 2000

Substitute Form PTO-1449 (Modified)	U.S. Department of Commerce Patent and Trademark Office	Attorney's Docket No. 06975-029006	Application No. 09/582297
Information Disclosure Statement by Applicant (Use several sheets if necessary) (37 CFR §1.98(b))		Applicant Kenneth Carbone et al.	
		Filing Date June 23, 2000	Group Art Unit

U.S. Patent Documents

Examiner Initial	Desig. ID	Patent Number	Issue Date	Patentee	Class	Subclass	Filing Date If Appropriate
	AA						
	AB						
	AC						
	AD						
	AE						
	AF						
	AG						
	AH						
	AI						
	AJ						
	AK						

Foreign Patent Documents or Published Foreign Patent Applications

Examiner Initial	Desig. ID	Document Number	Publication Date	Country or Patent Office	Class	Subclass	Translation Yes	Translation No
[illegible]	AL	0 862 304 A2	2/9/1998	Europe				
[illegible]	AM	WO 96/42145	12/27/96	PCT				
	AN							
	AO							
	AP							

Other Documents (include Author, Title, Date, and Place of Publication)

Examiner Initial	Desig. ID	Document
[illegible]	AQ	International Search Report, 25/6/1999; EPO
	AR	
	AS	
	AT	

Examiner Signature	Date Considered
[signature]	12/10/06

EXAMINER: Initials citation considered. Draw line through citation if not in conformance and not considered. Include copy of this form with next communication to applicant.

UNITED STATES PATENT AND TRADEMARK OFFICE

COMMISSIONER FOR PATENTS
UNITED STATES PATENT AND TRADEMARK OFFICE
WASHINGTON, D.C. 20231
www.uspto.gov

Bib Data Sheet

CONFIRMATION NO. 1661

SERIAL NUMBER	FILING DATE	CLASS	GROUP ART UNIT	ATTORNEY DOCKET NO.
09/582,297	04/08/2002 RULE	~~375~~ 709	2631	06975-029006

APPLICANTS

Kenneth Carbone, Annandale, VA;
Robert D. Greenlee, Leesburg, VA;
Marc A. Katchay, Silver Spring, MD;
Harry G. Morgan, Leesburg, VA;
Scott A. Quillen, Leesburg, VA;

**** CONTINUING DATA ***************************

THIS APPLICATION IS A 371 OF PCT/US98/27268 12/22/1998
WHICH CLAIMS BENEFIT OF 60/068,868 12/24/1997
AND CLAIMS BENEFIT OF 60/070,617 01/06/1998

**** FOREIGN APPLICATIONS **********************

None

Foreign Priority claimed	STATE OR COUNTRY	SHEETS DRAWING	TOTAL CLAIMS	INDEPENDENT CLAIMS
☐ yes ☒ no	VA	11	69	8

35 USC 119 (a-d) conditions met: ☐ yes ☒ no ☐ Met after Allowance

Verified and Acknowledged: Examiner's Signature _____ Initials _____

ADDRESS

Fish & Richardson
601 Thirteenth Street N W
Washington , DC 20005

TITLE

Asynchronous data protocol

FILING FEE RECEIVED	FEES: Authority has been given in Paper No. _____ to charge/credit DEPOSIT ACCOUNT No. _____ for following:	☐ All Fees ☐ 1.16 Fees (Filing) ☐ 1.17 Fees (Processing Ext. of time) ☐ 1.18 Fees (Issue) ☐ Other _____ ☐ Credit
2372		

Notice of References Cited	Application/Control No. 09/582,297	Applicant(s)/Patent Under Reexamination CARBONE ET AL.	
	Examiner Ramy M. Osman	Art Unit 2157	Page 1 of 1

U.S. PATENT DOCUMENTS

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Name	Classification
*	A	US-6,215,769 B1	04-2001	Ghani et al.	370/230
*	B	US-6,754,228 B1	06-2004	Ludwig, Reiner	370/468
*	C	US-6,038,601 A	03-2000	Lambert et al.	709/226
	D	US-			
	E	US-			
	F	US-			
	G	US-			
	H	US-			
	I	US-			
	J	US-			
	K	US-			
	L	US-			
	M	US-			

FOREIGN PATENT DOCUMENTS

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Country	Name	Classification
	N					
	O					
	P					
	Q					
	R					
	S					
	T					

NON-PATENT DOCUMENTS

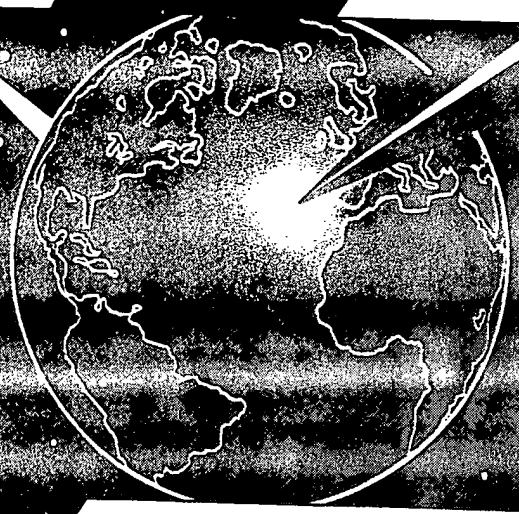
*		Include as applicable: Author, Title Date, Publisher, Edition or Volume, Pertinent Pages)
	U	Stevens, W. Richard, "TCP/IP Illustrated, Volume 1: The Protocols", Addison-Wesley, 1994 (ISBN-0-201-63346-9)
	V	
	W	
	X	

*A copy of this reference is not being furnished with this Office action. (See MPEP § 707.05(a).)
Dates in MM-YYYY format are publication dates. Classifications may be US or foreign.

TCP/IP Illustrated Volu

The P

W. R[illegible] tevens



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Networking, TCP/IP

TCP/IP Illustrated, Volume 1

"The word *illustrated* distinguishes this book from its many rivals. Stevens uses the Lawrence Berkeley Laboratories `tcpdump` program to capture packets in promiscuous mode under a variety of OS and TCP/IP implementations. Studying `tcpdump` output helps you understand how the various protocols work."

—Stan Kelly-Bootle, *Unix Review*

TCP/IP Illustrated is a complete and detailed guide to the entire TCP/IP protocol suite—with an important difference from other books on the subject. Rather than just describing what the RFCs say the protocol suite should do, this unique book uses a popular diagnostic tool so you may actually watch the protocols in action.

By forcing various conditions to occur—such as connection establishment, timeout and retransmission, and fragmentation—and then displaying the results, *TCP/IP Illustrated* gives you a much greater understanding of these concepts than words alone could provide. Whether you are new to TCP/IP or you have read other books on the subject, you will come away with an increased understanding of how and why TCP/IP works the way it does, as well as enhanced skill at developing applications that run over TCP/IP.

With this unique approach, *TCP/IP Illustrated* presents the structure and function of TCP/IP from the link layer up through the network, transport, and application layers. You will learn about the protocols that belong to each of these layers and how they operate under numerous implementations, including SunOS™ 4.1.3, Solaris® 2.2, UNIX® System V Release 4, BSD/386™, AIX® 3.2.2, and 4.4BSD.

In *TCP/IP Illustrated* you will find the most thorough coverage of TCP available—8 entire chapters. You will also find coverage of the newest TCP/IP features, including multicasting, path MTU discovery, and long fat pipes.

W. Richard Stevens is the highly-respected author of four other bestselling books, *TCP/IP Illustrated, Volume 2*—with Gary R. Wright (Addison-Wesley, 1995), *TCP/IP Illustrated, Volume 3* (Addison-Wesley, 1996), *Advanced Programming in the UNIX Environment* (Addison-Wesley, 1992), and the forthcoming *UNIX Network Programming, Second Edition* (Prentice-Hall, 1998). He is also a popular tutorials instructor and consultant.

Cover illustration by C. Shane Sykes
Text printed on recycled paper

ADDISON-WESLEY
Pearson Education

9 780201 633467 56999
ISBN 0-201-63346-9
$69.99 US
$108.99 CANADA

Pearson Education Corporate Sales Division
201 W. 103rd Street
Indianapolis, IN 46290
(800) 428-5331
corpsales@pearsoned.com

Visit AW on the Web: www.awl.com/cseng/

Library of Congress Cataloging-in-Publication Data
Stevens, W. Richard
TCP/IP Illustrated: the protocols/W. Richard Stevens.
p. cm.–(Addison-Wesley professional computing series)
Includes bibliographical references and index.
ISBN 0-201-63346-9 (v.1)
1.TCP/IP (Computer network protocol) I. Title. II. Series.
TK5105.55S74 1994
004.6'2—dc20

Text printed on recycled and acid-free paper.

ISBN 0201633469
23 2425262728 MA 06 05 04 03
23rd Printing September 2003

1

Introduction

1.1 Introduction

The TCP/IP protocol suite allows computers of all sizes, from many different computer vendors, running totally different operating systems, to communicate with each other. It is quite amazing because its use has far exceeded its original estimates. What started in the late 1960s as a government-financed research project into packet switching networks has, in the 1990s, turned into the most widely used form of networking between computers. It is truly an *open system* in that the definition of the protocol suite and many of its implementations are publicly available at little or no charge. It forms the basis for what is called the *worldwide Internet*, or the *Internet*, a wide area network (WAN) of more than one million computers that literally spans the globe.

This chapter provides an overview of the TCP/IP protocol suite, to establish an adequate background for the remaining chapters. For a historical perspective on the early development of TCP/IP see [Lynch 1993].

1.2 Layering

Networking *protocols* are normally developed in *layers*, with each layer responsible for a different facet of the communications. A *protocol suite*, such as TCP/IP, is the combination of different protocols at various layers. TCP/IP is normally considered to be a 4-layer system, as shown in Figure 1.1.

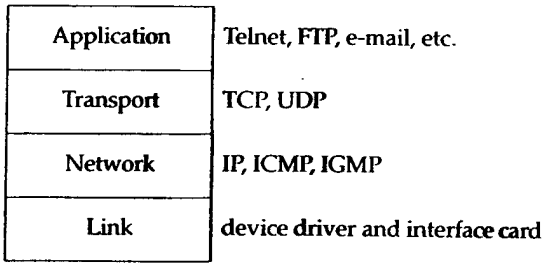


Figure 1.1 The four layers of the TCP/IP protocol suite.

Each layer has a different responsibility.

1. The *link* layer, sometimes called the *data-link* layer or *network interface* layer, normally includes the device driver in the operating system and the corresponding network interface card in the computer. Together they handle all the hardware details of physically interfacing with the cable (or whatever type of media is being used).

2. The *network* layer (sometimes called the *internet* layer) handles the movement of packets around the network. Routing of packets, for example, takes place here. IP (Internet Protocol), ICMP (Internet Control Message Protocol), and IGMP (Internet Group Management Protocol) provide the network layer in the TCP/IP protocol suite.

3. The *transport* layer provides a flow of data between two hosts, for the application layer above. In the TCP/IP protocol suite there are two vastly different transport protocols: TCP (Transmission Control Protocol) and UDP (User Datagram Protocol).

   TCP provides a reliable flow of data between two hosts. It is concerned with things such as dividing the data passed to it from the application into appropriately sized chunks for the network layer below, acknowledging received packets, setting timeouts to make certain the other end acknowledges packets that are sent, and so on. Because this reliable flow of data is provided by the transport layer, the application layer can ignore all these details.

   UDP, on the other hand, provides a much simpler service to the application layer. It just sends packets of data called *datagrams* from one host to the other, but there is no guarantee that the datagrams reach the other end. Any desired reliability must be added by the application layer.

   There is a use for each type of transport protocol, which we'll see when we look at the different applications that use TCP and UDP.

4. The *application* layer handles the details of the particular application. There are many common TCP/IP applications that almost every implementation provides:

   - Telnet for remote login,
   - FTP, the File Transfer Protocol,
   - SMTP, the Simple Mail Transfer protocol, for electronic mail,
   - SNMP, the Simple Network Management Protocol,

   and many more, some of which we cover in later chapters.

If we have two hosts on a local area network (LAN) such as an Ethernet, both running FTP, Figure 1.2 shows the protocols involved.

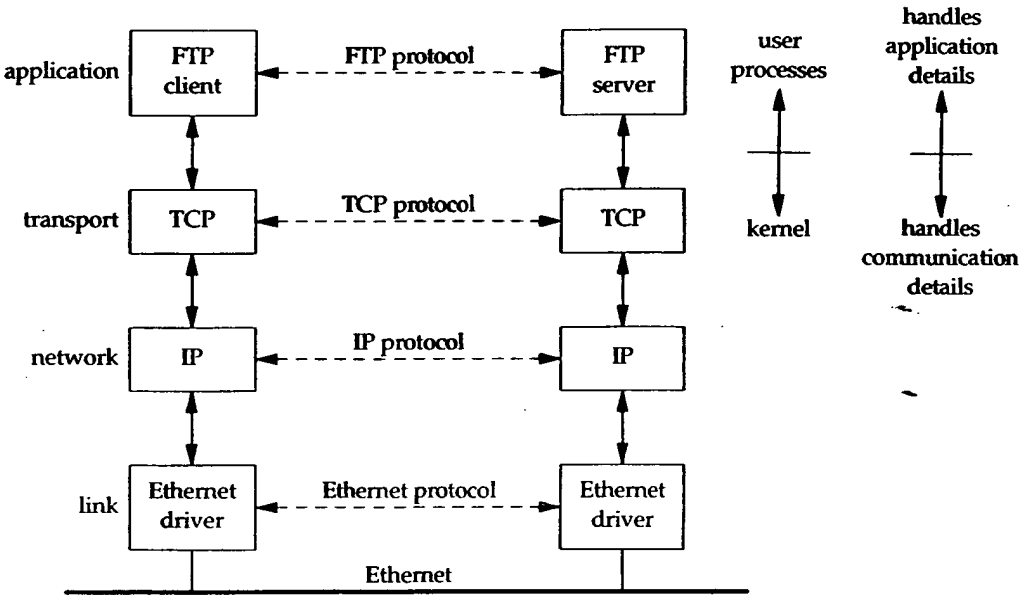


Figure 1.2 Two hosts on a LAN running FTP.

We have labeled one application box the FTP *client* and the other the FTP *server*. Most network applications are designed so that one end is the client and the other side the server. The server provides some type of service to clients, in this case access to files on the server host. In the remote login application, Telnet, the service provided to the client is the ability to login to the server's host.

Each layer has one or more protocols for communicating with its *peer* at the same layer. One protocol, for example, allows the two TCP layers to communicate, and another protocol lets the two IP layers communicate.

On the right side of Figure 1.2 we have noted that normally the application layer is a user process while the lower three layers are usually implemented in the kernel (the operating system). Although this isn't a requirement, it's typical and this is the way it's done under Unix.

There is another critical difference between the top layer in Figure 1.2 and the lower three layers. The application layer is concerned with the details of the application and not with the movement of data across the network. The lower three layers know nothing about the application but handle all the communication details.

We show four protocols in Figure 1.2, each at a different layer. FTP is an application layer protocol, TCP is a transport layer protocol, IP is a network layer protocol, and the Ethernet protocols operate at the link layer. The *TCP/IP protocol suite* is a combination of many protocols. Although the commonly used name for the entire protocol suite is TCP/IP, TCP and IP are only two of the protocols. (An alternative name is the *Internet Protocol Suite*.)

The purpose of the network interface layer and the application layer are obvious—the former handles the details of the communication media (Ethernet, token ring, etc.) while the latter handles one specific user application (FTP, Telnet, etc.). But on first glance the difference between the network layer and the transport layer is somewhat hazy. Why is there a distinction between the two? To understand the reason, we have to expand our perspective from a single network to a collection of networks.

One of the reasons for the phenomenal growth in networking during the 1980s was the realization that an island consisting of a stand-alone computer made little sense. A few stand-alone systems were collected together into a *network*. While this was progress, during the 1990s we have come to realize that this new, bigger island consisting of a single network doesn't make sense either. People are combining multiple networks together into an internetwork, or an *internet*. An internet is a collection of networks that all use the same protocol suite.

The easiest way to build an internet is to connect two or more networks with a *router*. This is often a special-purpose hardware box for connecting networks. The nice thing about routers is that they provide connections to many different types of physical networks: Ethernet, token ring, point-to-point links, FDDI (Fiber Distributed Data Interface), and so on.

> These boxes are also called *IP routers*, but we'll use the term *router*.
>
> Historically these boxes were called *gateways*, and this term is used throughout much of the TCP/IP literature. Today the term *gateway* is used for an application gateway: a process that connects two different protocol suites (say, TCP/IP and IBM's SNA) for one particular application (often electronic mail or file transfer).

Figure 1.3 shows an internet consisting of two networks: an Ethernet and a token ring, connected with a router. Although we show only two hosts communicating, with the router connecting the two networks, *any* host on the Ethernet can communicate with *any* host on the token ring.

In Figure 1.3 we can differentiate between an *end system* (the two hosts on either side) and an *intermediate system* (the router in the middle). The application layer and the transport layer use *end-to-end* protocols. In our picture these two layers are needed only on the end systems. The network layer, however, provides a *hop-by-hop* protocol and is used on the two end systems and every intermediate system.

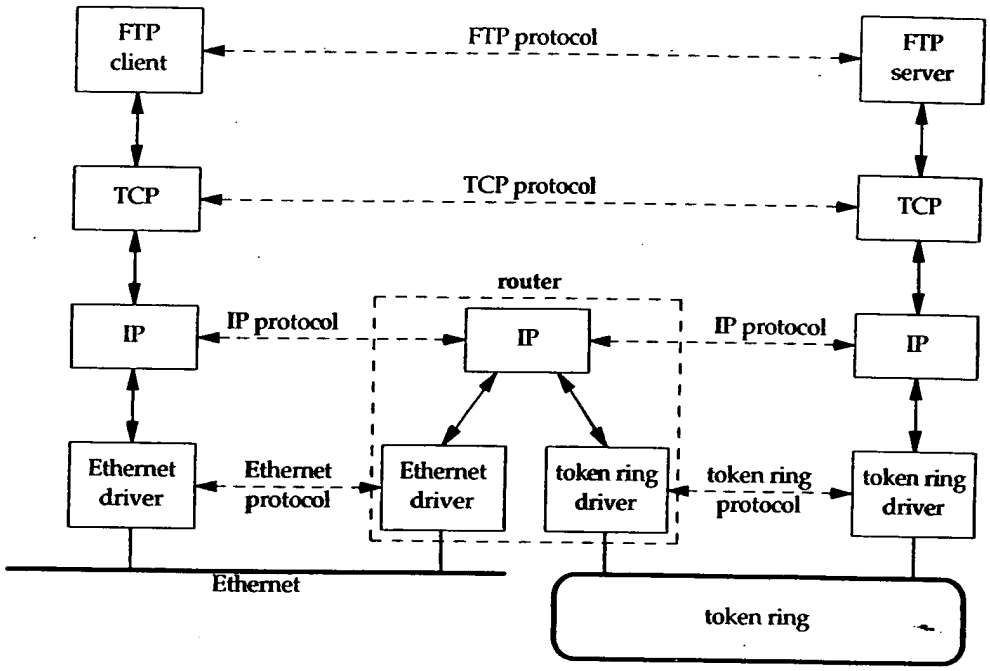


Figure 1.3 Two networks connected with a router.

In the TCP/IP protocol suite the network layer, IP, provides an unreliable service. That is, it does its best job of moving a packet from its source to its final destination, but there are no guarantees. TCP, on the other hand, provides a reliable transport layer using the unreliable service of IP. To provide this service, TCP performs timeout and retransmission, sends and receives end-to-end acknowledgments, and so on. The transport layer and the network layer have distinct responsibilities.

A router, by definition, has two or more network interface layers (since it connects two or more networks). Any system with multiple interfaces is called *multihomed*. A host can also be multihomed but unless it specifically forwards packets from one interface to another, it is not called a router. Also, routers need not be special hardware boxes that only move packets around an internet. Most TCP/IP implementations allow a multihomed host to act as a router also, but the host needs to be specifically configured for this to happen. In this case we can call the system either a host (when an application such as FTP or Telnet is being used) or a router (when it's forwarding packets from one network to another). We'll use whichever term makes sense given the context.

One of the goals of an internet is to hide all the details of the physical layout of the internet from the applications. Although this isn't obvious from our two-network internet in Figure 1.3, the application layers can't care (and don't care) that one host is on an Ethernet, the other on a token ring, with a router between. There could be 20 routers between, with additional types of physical interconnections, and the applications would run the same. This hiding of the details is what makes the concept of an internet so powerful and useful.

Another way to connect networks is with a *bridge*. These connect networks at the link layer, while routers connect networks at the network layer. Bridges makes multiple LANs appear to the upper layers as a single LAN.

TCP/IP internets tend to be built using routers instead of bridges, so we'll focus on routers. Chapter 12 of [Perlman 1992] compares routers and bridges.

1.3 TCP/IP Layering

There are more protocols in the TCP/IP protocol suite. Figure 1.4 shows some of the additional protocols that we talk about in this text.

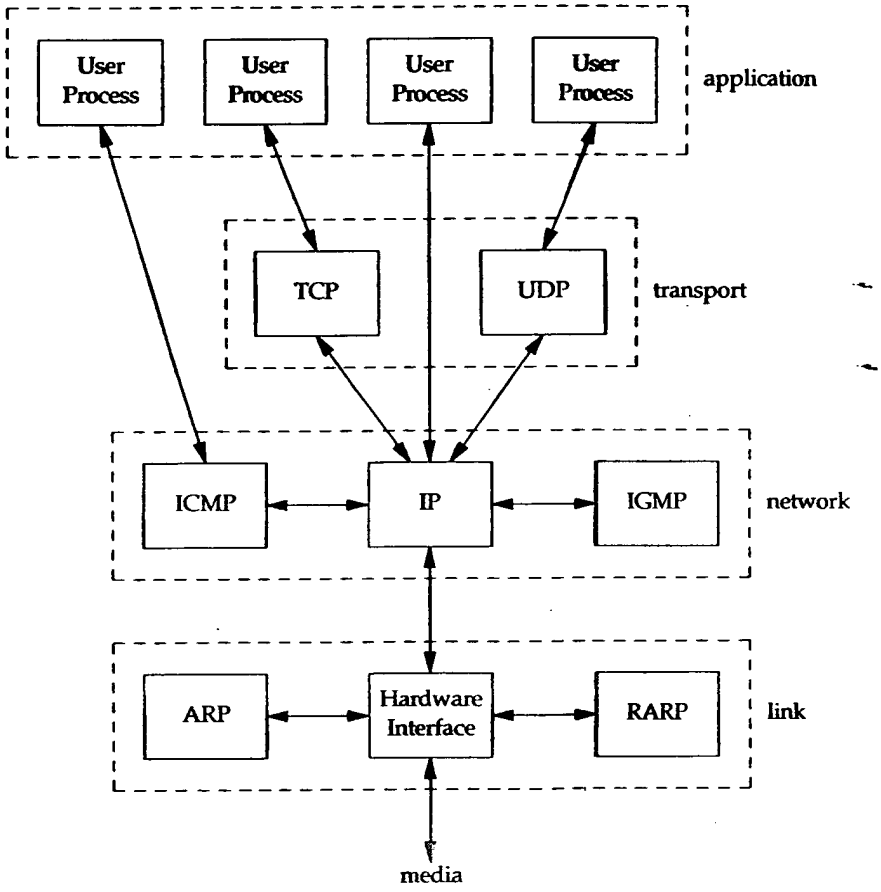


Figure 1.4 Various protocols at the different layers in the TCP/IP protocol suite.

TCP and UDP are the two predominant transport layer protocols. Both use IP as the network layer.

TCP provides a reliable transport layer, even though the service it uses (IP) is unreliable. Chapters 17 through 22 provide a detailed look at the operation of TCP. We then look at some TCP applications: Telnet and Rlogin in Chapter 26, FTP in Chapter 27, and SMTP in Chapter 28. The applications are normally user processes.

UDP sends and receives *datagrams* for applications. A datagram is a unit of information (i.e., a certain number of bytes of information that is specified by the sender) that travels from the sender to the receiver. Unlike TCP, however, UDP is unreliable. There is no guarantee that the datagram ever gets to its final destination. Chapter 11 looks at UDP, and then Chapter 14 (the Domain Name System), Chapter 15 (the Trivial File Transfer Protocol), and Chapter 16 (the Bootstrap Protocol) look at some applications that use UDP. SNMP (the Simple Network Management Protocol) also uses UDP, but since it deals with many of the other protocols, we save a discussion of it until Chapter 25.

IP is the main protocol at the network layer. It is used by both TCP and UDP. Every piece of TCP and UDP data that gets transferred around an internet goes through the IP layer at both end systems and at every intermediate router. In Figure 1.4 we also show an application accessing IP directly. This is rare, but possible. (Some older routing protocols were implemented this way. Also, it is possible to experiment with new transport layer protocols using this feature.) Chapter 3 looks at IP, but we save some of the details for later chapters where their discussion makes more sense. Chapters 9 and 10 look at how IP performs routing.

ICMP is an adjunct to IP. It is used by the IP layer to exchange error messages and other vital information with the IP layer in another host or router. Chapter 6 looks at ICMP in more detail. Although ICMP is used primarily by IP, it is possible for an application to also access it. Indeed we'll see that two popular diagnostic tools, Ping and Traceroute (Chapters 7 and 8), both use ICMP.

IGMP is the Internet Group Management Protocol. It is used with multicasting: sending a UDP datagram to multiple hosts. We describe the general properties of broadcasting (sending a UDP datagram to every host on a specified network) and multicasting in Chapter 12, and then describe IGMP itself in Chapter 13.

ARP (Address Resolution Protocol) and RARP (Reverse Address Resolution Protocol) are specialized protocols used only with certain types of network interfaces (such as Ethernet and token ring) to convert between the addresses used by the IP layer and the addresses used by the network interface. We examine these protocols in Chapters 4 and 5, respectively.

1.4 Internet Addresses

Every interface on an internet must have a unique *Internet address* (also called an *IP address*). These addresses are 32-bit numbers. Instead of using a flat address space such as 1, 2, 3, and so on, there is a structure to Internet addresses. Figure 1.5 shows the five different classes of Internet addresses.

These 32-bit addresses are normally written as four decimal numbers, one for each byte of the address. This is called *dotted-decimal* notation. For example, the class B address of the author's primary system is 140.252.13.33.

The easiest way to differentiate between the different classes of addresses is to look at the first number of a dotted-decimal address. Figure 1.6 shows the different classes, with the first number in boldface.

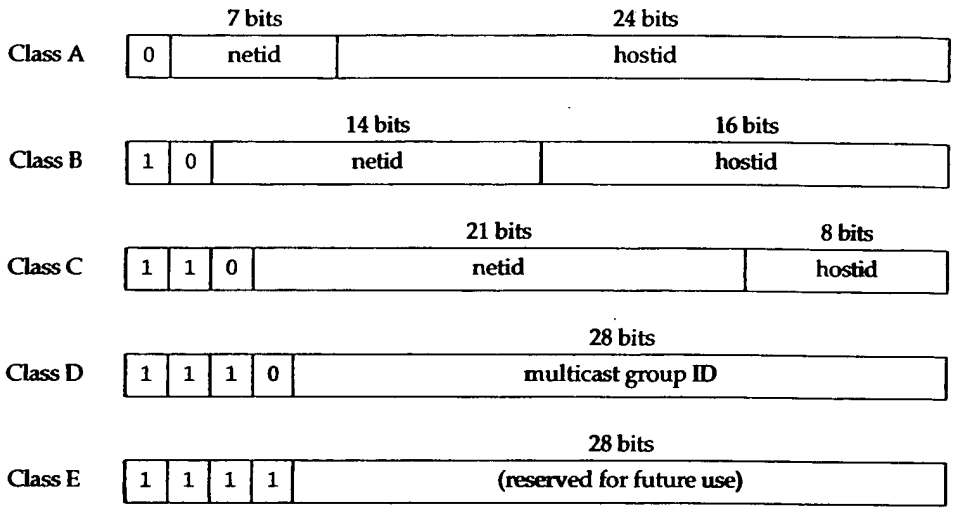


Figure 1.5 The five different classes of Internet addresses.

Class	Range
A	0.0.0.0 to 127.255.255.255
B	128.0.0.0 to 191.255.255.255
C	192.0.0.0 to 223.255.255.255
D	224.0.0.0 to 239.255.255.255
E	240.0.0.0 to 255.255.255.255

Figure 1.6 Ranges for different classes of IP addresses.

It is worth reiterating that a multihomed host will have multiple IP addresses: one per interface.

Since every interface on an internet must have a unique IP address, there must be one central authority for allocating these addresses for networks connected to the worldwide Internet. That authority is the *Internet Network Information Center*, called the InterNIC. The InterNIC assigns only network IDs. The assignment of host IDs is up to the system administrator.

> Registration services for the Internet (IP addresses and DNS domain names) used to be handled by the NIC, at `nic.ddn.mil`. On April 1, 1993, the InterNIC was created. Now the NIC handles these requests only for the *Defense Data Network* (DDN). All other Internet users now use the InterNIC registration services, at `rs.internic.net`.
>
> There are actually three parts to the InterNIC: registration services (`rs.internic.net`), directory and database services (`ds.internic.net`), and information services (`is.internic.net`). See Exercise 1.8 for additional information on the InterNIC.

There are three types of IP addresses: *unicast* (destined for a single host), *broadcast* (destined for all hosts on a given network), and *multicast* (destined for a set of hosts that belong to a multicast group). Chapters 12 and 13 look at broadcasting and multicasting in more detail.

In Section 3.4 we'll extend our description of IP addresses to include subnetting, after describing IP routing. Figure 3.9 shows the special case IP addresses: host IDs and network IDs of all zero bits or all one bits.

1.5 The Domain Name System

Although the network interfaces on a host, and therefore the host itself, are known by IP addresses, humans work best using the *name* of a host. In the TCP/IP world the *Domain Name System* (DNS) is a distributed database that provides the mapping between IP addresses and hostnames. Chapter 14 looks into the DNS in detail.

For now we must be aware that any application can call a standard library function to look up the IP address (or addresses) corresponding to a given hostname. Similarly a function is provided to do the reverse lookup—given an IP address, look up the corresponding hostname.

Most applications that take a hostname as an argument also take an IP address. When we use the Telnet client in Chapter 4, for example, one time we specify a hostname and another time we specify an IP address.

1.6 Encapsulation

When an application sends data using TCP, the data is sent down the protocol stack, through each layer, until it is sent as a stream of bits across the network. Each layer adds information to the data by prepending headers (and sometimes adding trailer information) to the data that it receives. Figure 1.7 shows this process. The unit of data that TCP sends to IP is called a *TCP segment*. The unit of data that IP sends to the network interface is called an *IP datagram*. The stream of bits that flows across the Ethernet is called a *frame*.

The numbers at the bottom of the headers and trailer of the Ethernet frame in Figure 1.7 are the typical sizes of the headers in bytes. We'll have more to say about each of these headers in later sections.

A physical property of an Ethernet frame is that the size of its data must be between 46 and 1500 bytes. We'll encounter this minimum in Section 4.5 and we cover the maximum in Section 2.8.

> All the Internet standards and most books on TCP/IP use the term *octet* instead of byte. The use of this cute, but baroque term is historical, since much of the early work on TCP/IP was done on systems such as the DEC-10, which did not use 8-bit bytes. Since almost every current computer system uses 8-bit bytes, we'll use the term *byte* in this text.
>
> To be completely accurate in Figure 1.7 we should say that the unit of data passed between IP and the network interface is a *packet*. This packet can be either an IP datagram or a fragment of an IP datagram. We discuss fragmentation in detail in Section 11.5.

We could draw a nearly identical picture for UDP data. The only changes are that the unit of information that UDP passes to IP is called a *UDP datagram*, and the size of the UDP header is 8 bytes.

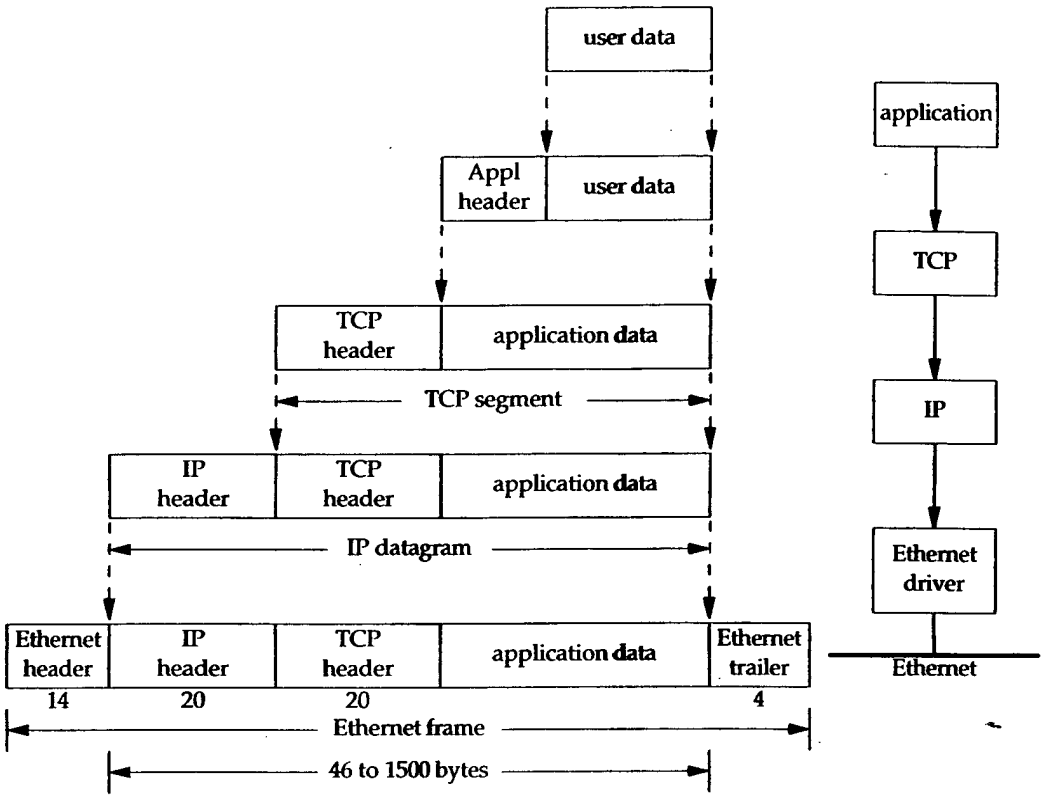


Figure 1.7 Encapsulation of data as it goes down the protocol stack.

Recall from Figure 1.4 (p. 6) that TCP, UDP, ICMP, and IGMP all send data to IP. IP must add some type of identifier to the IP header that it generates, to indicate the layer to which the data belongs. IP handles this by storing an 8-bit value in its header called the *protocol* field. A value of 1 is for ICMP, 2 is for IGMP, 6 indicates TCP, and 17 is for UDP.

Similarly, many different applications can be using TCP or UDP at any one time. The transport layer protocols store an identifier in the headers they generate to identify the application. Both TCP and UDP use 16-bit *port numbers* to identify applications. TCP and UDP store the source port number and the destination port number in their respective headers.

The network interface sends and receives frames on behalf of IP, ARP, and RARP. There must be some form of identification in the Ethernet header indicating which network layer protocol generated the data. To handle this there is a 16-bit frame type field in the Ethernet header.

1.7 Demultiplexing

When an Ethernet frame is received at the destination host it starts its way up the protocol stack and all the headers are removed by the appropriate protocol box. Each protocol box looks at certain identifiers in its header to determine which box in the next upper layer receives the data. This is called *demultiplexing*. Figure 1.8 shows how this takes place.

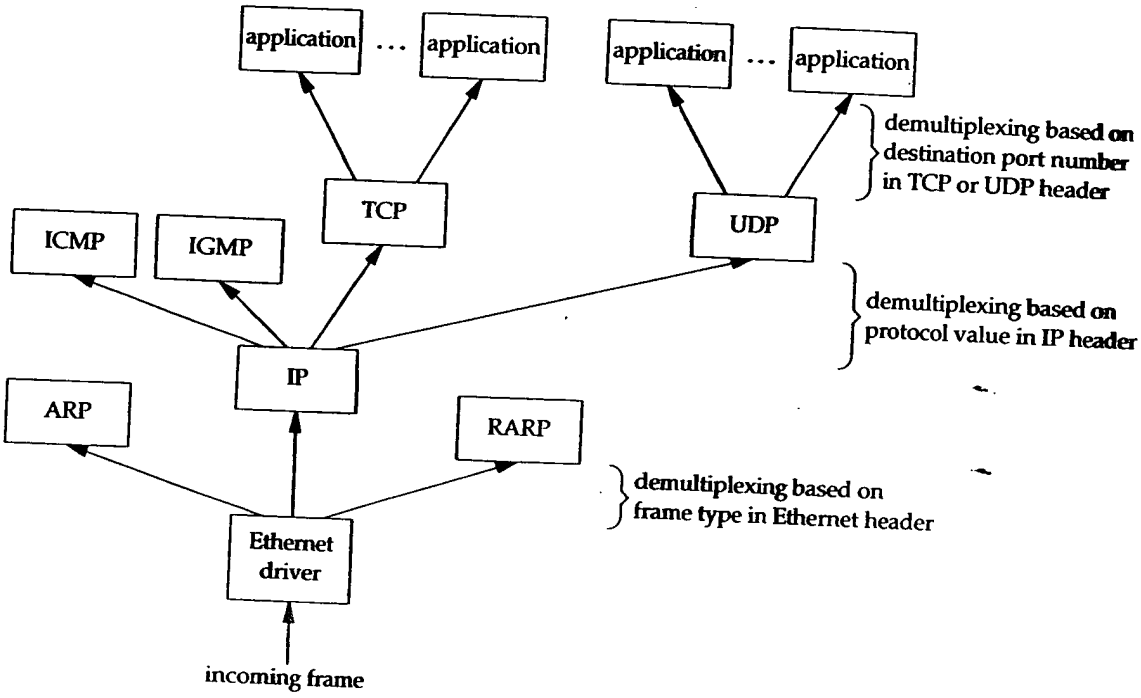


Figure 1.8 The demultiplexing of a received Ethernet frame.

Positioning the protocol boxes labeled "ICMP" and "IGMP" is always a challenge. In Figure 1.4 we showed them at the same layer as IP, because they really are adjuncts to IP. But here we show them above IP, to reiterate that ICMP messages and IGMP messages are encapsulated in IP datagrams.

We have a similar problem with the boxes "ARP" and "RARP." Here we show them above the Ethernet device driver because they both have their own Ethernet frame types, like IP datagrams. But in Figure 2.4 we'll show ARP as part of the Ethernet device driver, beneath IP, because that's where it logically fits.

Realize that these pictures of layered protocol boxes are not perfect.

When we describe TCP in detail we'll see that it really demultiplexes incoming segments using the destination port number, the source IP address, and the source port number.

1.8 Client–Server Model

Most networking applications are written assuming one side is the client and the other the server. The purpose of the application is for the server to provide some defined service for clients.

We can categorize servers into two classes: iterative or concurrent. An *iterative server* iterates through the following steps.

I1. Wait for a client request to arrive.

I2. Process the client request.

I3. Send the response back to the client that sent the request.

I4. Go back to step I1.

The problem with an iterative server is when step I2 takes a while. During this time no other clients are serviced.

A *concurrent server*, on the other hand, performs the following steps.

C1. Wait for a client request to arrive.

C2. Start a new server to handle this client's request. This may involve creating a new process, task, or thread, depending on what the underlying operating system supports. How this step is performed depends on the operating system.

This new server handles this client's entire request. When complete, this new server terminates.

C3. Go back to step C1.

The advantage of a concurrent server is that the server just spawns other servers to handle the client requests. Each client has, in essence, its own server. Assuming the operating system allows multiprogramming, multiple clients are serviced concurrently.

The reason we categorize servers, and not clients, is because a client normally can't tell whether it's talking to an iterative server or a concurrent server.

As a general rule, TCP servers are concurrent, and UDP servers are iterative, but there are a few exceptions. We'll look in detail at the impact of UDP on its servers in Section 11.12, and the impact of TCP on its servers in Section 18.11.

1.9 Port Numbers

We said that TCP and UDP identify applications using 16-bit port numbers. How are these port numbers chosen?

Servers are normally known by their *well-known* port number. For example, every TCP/IP implementation that provides an FTP server provides that service on TCP port

21. Every Telnet server is on TCP port 23. Every implementation of TFTP (the Trivial File Transfer Protocol) is on UDP port 69. Those services that can be provided by any implementation of TCP/IP have well-known port numbers between 1 and 1023. The well-known ports are managed by the *Internet Assigned Numbers Authority* (IANA).

> Until 1992 the well-known ports were between 1 and 255. Ports between 256 and 1023 were normally used by Unix systems for Unix-specific services—that is, services found on a Unix system, but probably not found on other operating systems. The IANA now manages the ports between 1 and 1023.
>
> An example of the difference between an Internet-wide service and a Unix-specific service is the difference between Telnet and Rlogin. Both allow us to login across a network to another host. Telnet is a TCP/IP standard with a well-known port number of 23 and can be implemented on almost any operating system. Rlogin, on the other hand, was originally designed for Unix systems (although many non-Unix systems now provide it also) so its well-known port was chosen in the early 1980s as 513.

A client usually doesn't care what port number it uses on its end. All it needs to be certain of is that whatever port number it uses be unique on its host. Client port numbers are called *ephemeral ports* (i.e., short lived). This is because a client typically exists only as long as the user running the client needs its service, while servers typically run as long as the host is up.

Most TCP/IP implementations allocate ephemeral port numbers between 1024 and 5000. The port numbers above 5000 are intended for other servers (those that aren't well known across the Internet). We'll see many examples of how ephemeral ports are allocated in the examples throughout the text.

> Solaris 2.2 is a notable exception. By default the ephemeral ports for TCP and UDP start at 32768. Section E.4 details the configuration options that can be modified by the system administrator to change these defaults.

The well-known port numbers are contained in the file `/etc/services` on most Unix systems. To find the port numbers for the Telnet server and the Domain Name System, we can execute

```
sun % grep telnet /etc/services
telnet    23/tcp                    says it uses TCP port 23

sun % grep domain /etc/services
domain    53/udp                    says it uses UDP port 53
domain    53/tcp                    and TCP port 53
```

Reserved Ports

Unix systems have the concept of *reserved ports*. Only a process with superuser privileges can assign itself a reserved port.

These port numbers are in the range of 1 to 1023, and are used by some applications (notably Rlogin, Section 26.2), as part of the authentication between the client and server.

1.10 Standardization Process

Who controls the TCP/IP protocol suite, approves new standards, and the like? There are four groups responsible for Internet technology.

1. The *Internet Society* (ISOC) is a professional society to facilitate, support, and promote the evolution and growth of the Internet as a global research communications infrastructure.
2. The *Internet Architecture Board* (IAB) is the technical oversight and coordination body. It is composed of about 15 international volunteers from various disciplines and serves as the final editorial and technical review board for the quality of Internet standards. The IAB falls under the ISOC.
3. The *Internet Engineering Task Force* (IETF) is the near-term, standards-oriented group, divided into nine areas (applications, routing and addressing, security, etc.). The IETF develops the specifications that become Internet standards. An additional *Internet Engineering Steering Group* (IESG) was formed to help the IETF chair.
4. The *Internet Research Task Force* (IRTF) pursues long-term research projects.

Both the IRTF and the IETF fall under the IAB. [Crocker 1993] provides additional details on the standardization process within the Internet, as well as some of its early history.

1.11 RFCs

All the official standards in the internet community are published as a *Request for Comment*, or *RFC*. Additionally there are lots of RFCs that are not official standards, but are published for informational purposes. The RFCs range in size from 1 page to almost 200 pages. Each is identified by a number, such as RFC 1122, with higher numbers for newer RFCs.

All the RFCs are available at no charge through electronic mail or using FTP across the Internet. Sending electronic mail as shown here:

```
To: rfc-info@ISI.EDU
Subject: getting rfcs

help: ways_to_get_rfcs
```

returns a detailed listing of various ways to obtain the RFCs.

The latest RFC index is always a starting point when looking for something. This index specifies when a certain RFC has been replaced by a newer RFC, and if a newer RFC updates some of the information in that RFC.

There are a few important RFCs.

1. The *Assigned Numbers RFC* specifies all the magic numbers and constants that are used in the Internet protocols. At the time of this writing the latest version

of this RFC is 1340 [Reynolds and Postel 1992]. All the Internet-wide well-known ports are listed here.

When this RFC is updated (it is normally updated at least yearly) the index listing for 1340 will indicate which RFC has replaced it.

2. The *Internet Official Protocol Standards*, currently RFC 1600 [Postel 1994]. This RFC specifies the state of standardization of the various Internet protocols. Each protocol has one of the following states of standardization: standard, draft standard, proposed standard, experimental, informational, or historic. Additionally each protocol has a requirement level: required, recommended, elective, limited use, or not recommended.

   Like the Assigned Numbers RFC, this RFC is also reissued regularly. Be sure you're reading the current copy.

3. The *Host Requirements RFCs*, 1122 and 1123 [Braden 1989a, 1989b]. RFC 1122 handles the link layer, network layer, and transport layer, while RFC 1123 handles the application layer. These two RFCs make numerous corrections and interpretations of the important earlier RFCs, and are often the starting point when looking at any of the finer details of a given protocol. They list the features and implementation details of the protocols as either "must," "should," "may," "should not," or "must not."

   [Borman 1993b] provides a practical look at these two RFCs, and RFC 1127 [Braden 1989c] provides an informal summary of the discussions and conclusions of the working group that developed the Host Requirements RFCs.

4. The *Router Requirements RFC*. The official version of this is RFC 1009 [Braden and Postel 1987], but a new version is nearing completion [Almquist 1993]. This is similar to the host requirements RFCs, but specifies the unique requirements of routers.

1.12 Standard, Simple Services

There are a few standard, simple services that almost every implementation provides. We'll use some of these servers throughout the text, usually with the Telnet client. Figure 1.9 describes these services. We can see from this figure that when the same service is provided using both TCP and UDP, both port numbers are normally chosen to be the same.

> If we examine the port numbers for these standard services and other standard TCP/IP services (Telnet, FTP, SMTP, etc.), most are odd numbers. This is historical as these port numbers are derived from the NCP port numbers. (NCP, the Network Control Protocol, preceded TCP as a transport layer protocol for the ARPANET.) NCP was simplex, not full-duplex, so each application required two connections, and an even–odd pair of port numbers was reserved for each application. When TCP and UDP became the standard transport layers, only a single port number was needed per application, so the odd port numbers from NCP were used.

Name	TCP port	UDP port	RFC	Description
echo	7	7	862	Server returns whatever the client sends.
discard	9	9	863	Server discards whatever the client sends.
daytime	13	13	867	Server returns the time and date in a human-readable format.
chargen	19	19	864	TCP server sends a continual stream of characters, until the connection is terminated by the client. UDP server sends a datagram containing a random number of characters each time the client sends a datagram.
time	37	37	868	Server returns the time as a 32-bit binary number. This number represents the number of seconds since midnight January 1, 1900, UTC.

Figure 1.9 Standard, simple services provided by most implementations.

1.13 The Internet

In Figure 1.3 we showed an *internet* composed of two networks—an Ethernet and a token ring. In Sections 1.4 and 1.9 we talked about the worldwide *Internet* and the need to allocate IP addresses centrally (the InterNIC) and the well-known port numbers (the IANA). The word *internet* means different things depending on whether it's capitalized or not.

The lowercase *internet* means multiple networks connected together, using a common protocol suite. The uppercase *Internet* refers to the collection of hosts (over one million) around the world that can communicate with each other using TCP/IP. While the Internet is an internet, the reverse is not true.

1.14 Implementations

The de facto standard for TCP/IP implementations is the one from the Computer Systems Research Group at the University of California at Berkeley. Historically this has been distributed with the 4.x BSD system (Berkeley Software Distribution), and with the "BSD Networking Releases." This source code has been the starting point for many other implementations.

Figure 1.10 shows a chronology of the various BSD releases, indicating the important TCP/IP features. The BSD Networking Releases shown on the left side are publicly available source code releases containing all of the networking code: both the protocols themselves and many of the applications and utilities (such as Telnet and FTP).

Throughout the text we'll use the term *Berkeley-derived implementation* to refer to vendor implementations such as SunOS 4.x, SVR4, and AIX 3.2 that were originally developed from the Berkeley sources. These implementations have much in common, often including the same bugs!

2.6 PPP: Point-to-Point Protocol

PPP, the Point-to-Point Protocol, corrects all the deficiencies in SLIP. PPP consists of three components.

1. A way to encapsulate IP datagrams on a serial link. PPP supports either an asynchronous link with 8 bits of data and no parity (i.e., the ubiquitous serial interface found on most computers) or bit-oriented synchronous links.
2. A *link control protocol* (LCP) to establish, configure, and test the data-link connection. This allows each end to negotiate various options.
3. A family of *network control protocols* (NCPs) specific to different network layer protocols. RFCs currently exist for IP, the OSI network layer, DECnet, and AppleTalk. The IP NCP, for example, allows each end to specify if it can perform header compression, similar to CSLIP. (The acronym NCP was also used for the predecessor to TCP.)

RFC 1548 [Simpson 1993] specifies the encapsulation method and the link control protocol. RFC 1332 [McGregor 1992] specifies the network control protocol for IP.

The format of the PPP frames was chosen to look like the ISO HDLC standard (high-level data link control). Figure 2.3 shows the format of PPP frames.

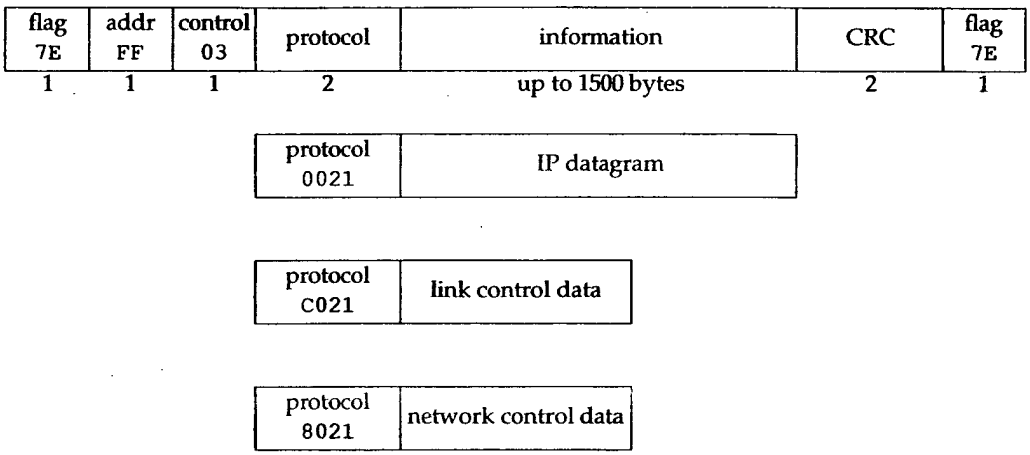


Figure 2.3 Format of PPP frames.

Each frame begins and ends with a *flag* byte whose value is `0x7e`. This is followed by an *address* byte whose value is always `0xff`, and then a *control* byte, with a value of `0x03`.

Next comes the *protocol* field, similar in function to the Ethernet *type* field. A value of `0x0021` means the *information* field is an IP datagram, a value of `0xc021` means the *information* field is link control data, and a value of `0x8021` is for network control data.

The *CRC* field (or FCS, for frame check sequence) is a cyclic redundancy check, to detect errors in the frame.

Since the byte value `0x7e` is the *flag* character, PPP needs to escape this byte when it appears in the *information* field. On a synchronous link this is done by the hardware using a technique called *bit stuffing* [Tanenbaum 1989]. On asynchronous links the special byte `0x7d` is used as an escape character. Whenever this escape character appears in a PPP frame, the next character in the frame has had its sixth bit complemented, as follows:

1. The byte `0x7e` is transmitted as the 2-byte sequence `0x7d, 0x5e`. This is the escape of the *flag* byte.
2. The byte `0x7d` is transmitted as the 2-byte sequence `0x7d, 0x5d`. This is the escape of the escape byte.
3. By default, a byte with a value less than `0x20` (i.e., an ASCII control character) is also escaped. For example, the byte `0x01` is transmitted as the 2-byte sequence `0x7d, 0x21`. (In this case the complement of the sixth bit turns the bit on, whereas in the two previous examples the complement turned the bit off.)

   The reason for doing this is to prevent these bytes from appearing as ASCII control characters to the serial driver on either host, or to the modems, which sometimes interpret these control characters specially. It is also possible to use the link control protocol to specify which, if any, of these 32 values must be escaped. By default, all 32 are escaped.

Since PPP, like SLIP, is often used across slow serial links, reducing the number of bytes per frame reduces the latency for interactive applications. Using the link control protocol, most implementations negotiate to omit the constant *address* and *control* fields and to reduce the size of the *protocol* field from 2 bytes to 1 byte. If we then compare the framing overhead in a PPP frame, versus the 2-byte framing overhead in a SLIP frame (Figure 2.2), we see that PPP adds three additional bytes: 1 byte for the *protocol* field, and 2 bytes for the *CRC*. Additionally, using the IP network control protocol, most implementations then negotiate to use Van Jacobson header compression (identical to CSLIP compression) to reduce the size of the IP and TCP headers.

In summary, PPP provides the following advantages over SLIP: (1) support for multiple protocols on a single serial line, not just IP datagrams, (2) a cyclic redundancy check on every frame, (3) dynamic negotiation of the IP address for each end (using the IP network control protocol), (4) TCP and IP header compression similar to CSLIP, and (5) a link control protocol for negotiating many data-link options. The price we pay for all these features is 3 bytes of additional overhead per frame, a few frames of negotiation when the link is established, and a more complex implementation.

> Despite all the added benefits of PPP over SLIP, today there are more SLIP users than PPP users. As implementations become more widely available, and as vendors start to support PPP, it should (eventually) replace SLIP.

2.7 Loopback Interface

Most implementations support a *loopback interface* that allows a client and server on the same host to communicate with each other using TCP/IP. The class A network ID 127 is reserved for the loopback interface. By convention, most systems assign the IP address of 127.0.0.1 to this interface and assign it the name `localhost`. An IP datagram sent to the loopback interface must not appear on any network.

Although we could imagine the transport layer detecting that the other end is the loopback address, and short circuiting some of the transport layer logic and all of the network layer logic, most implementations perform complete processing of the data in the transport layer and network layer, and only loop the IP datagram back to itself when the datagram leaves the bottom of the network layer.

Figure 2.4 shows a simplified diagram of how the loopback interface processes IP datagrams.

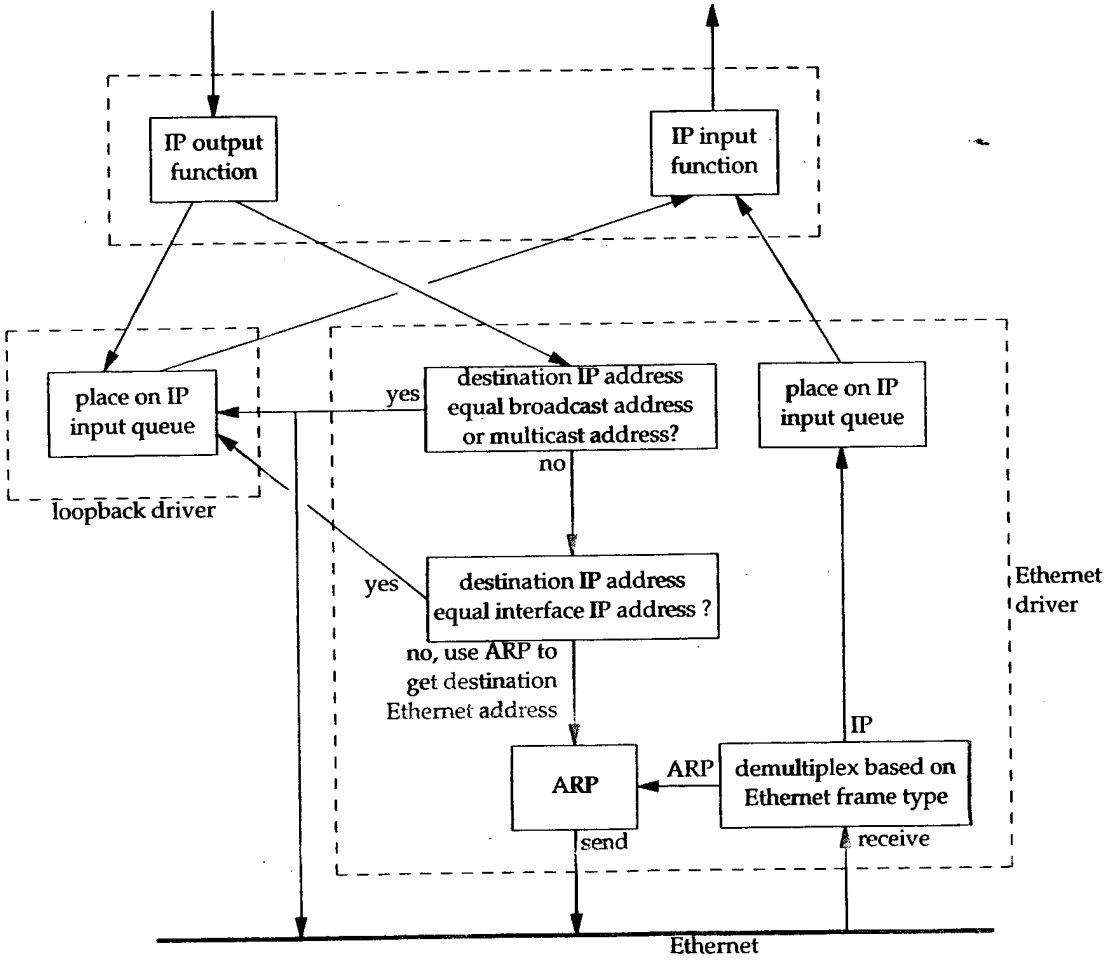


Figure 2.4 Processing of IP datagrams by loopback interface.

The key points to note in this figure are as follows:

1. Everything sent to the loopback address (normally 127.0.0.1) appears as IP input.
2. Datagrams sent to a broadcast address or a multicast address are copied to the loopback interface and sent out on the Ethernet. This is because the definition of broadcasting and multicasting (Chapter 12) includes the sending host.
3. Anything sent to one of the host's own IP addresses is sent to the loopback interface.

While it may seem inefficient to perform all the transport layer and IP layer processing of the loopback data, it simplifies the design because the loopback interface appears as just another link layer to the network layer. The network layer passes a datagram to the loopback interface like any other link layer, and it happens that the loopback interface then puts the datagram back onto IP's input queue.

Another implication of Figure 2.4 is that IP datagrams sent to the one of the host's own IP addresses normally do not appear on the corresponding network. For example, on an Ethernet, normally the packet is not transmitted and then read back. Comments in some of the BSD Ethernet device drivers indicate that many Ethernet interface cards are not capable of reading their own transmissions. Since a host must process IP datagrams that it sends to itself, handling these packets as shown in Figure 2.4 is the simplest way to accomplish this.

> The 4.4BSD implementation defines the variable `useloopback` and initializes it to 1. If this variable is set to 0, however, the Ethernet driver sends local packets onto the network instead of sending them to the loopback driver. This may or may not work, depending on your Ethernet interface card and device driver.

2.8 MTU

As we can see from Figure 2.1, there is a limit on the size of the frame for both Ethernet encapsulation and 802.3 encapsulation. This limits the number of bytes of data to 1500 and 1492, respectively. This characteristic of the link layer is called the *MTU*, its maximum transmission unit. Most types of networks have an upper limit.

If IP has a datagram to send, and the datagram is larger than the link layer's MTU, IP performs *fragmentation*, breaking the datagram up into smaller pieces (fragments), so that each fragment is smaller than the MTU. We discuss IP fragmentation in Section 11.5.

Figure 2.5 lists some typical MTU values, taken from RFC 1191 [Mogul and Deering 1990]. The listed MTU for a point-to-point link (e.g., SLIP or PPP) is not a physical characteristic of the network media. Instead it is a logical limit to provide adequate response time for interactive use. In the Section 2.10 we'll see where this limit comes from.

In Section 3.9 we'll use the `netstat` command to print the MTU of an interface.

Network	MTU (bytes)
Hyperchannel	65535
16 Mbits/sec token ring (IBM)	17914
4 Mbits/sec token ring (IEEE 802.5)	4464
FDDI	4352
Ethernet	1500
IEEE 802.3/802.2	1492
X.25	576
Point-to-point (low delay)	296

Figure 2.5 Typical maximum transmission units (MTUs).

2.9 Path MTU

When two hosts on the same network are communicating with each other, it is the MTU of the network that is important. But when two hosts are communicating across multiple networks, each link can have a different MTU. The important numbers are not the MTUs of the two networks to which the two hosts connect, but rather the smallest MTU of any data link that packets traverse between the two hosts. This is called the *path MTU*.

The path MTU between any two hosts need not be constant. It depends on the route being used at any time. Also, routing need not be symmetric (the route from A to B may not be the reverse of the route from B to A), hence the path MTU need not be the same in the two directions.

RFC 1191 [Mogul and Deering 1990] specifies the "path MTU discovery mechanism," a way to determine the path MTU at any time. We'll see how this mechanism operates after we've described ICMP and IP fragmentation. In Section 11.6 we'll examine the ICMP unreachable error that is used with this discovery mechanism and in Section 11.7 we'll show a version of the `traceroute` program that uses this mechanism to determine the path MTU to a destination. Sections 11.8 and 24.2 show how UDP and TCP operate when the implementation supports path MTU discovery.

2.10 Serial Line Throughput Calculations

If the line speed is 9600 bits/sec, with 8 bits per byte, plus 1 start bit and 1 stop bit, the line speed is 960 bytes/sec. Transferring a 1024-byte packet at this speed takes 1066 ms. If we're using the SLIP link for an interactive application, along with an application such as FTP that sends or receives 1024-byte packets, we have to wait, on the average, half of this time (533 ms) to send our interactive packet.

This assumes that our interactive packet will be sent across the link before any further "big" packets. Most SLIP implementations do provide this type-of-service queueing, placing interactive traffic ahead of bulk data traffic. The interactive traffic is normally Telnet, Rlogin, and the control portion (the user commands, not the data) of FTP.

This type of service queueing is imperfect. It cannot affect noninteractive traffic that is already queued downstream (e.g., at the serial driver). Also newer modems have large buffers so noninteractive traffic may already be buffered in the modem.

Waiting 533 ms is unacceptable for interactive response. Human factors studies have found that an interactive response time longer than 100–200 ms is perceived as bad [Jacobson 1990a]. This is the round-trip time for an interactive packet to be sent and something to be returned (normally a character echo).

Reducing the MTU of the SLIP link to 256 means the maximum amount of time the link can be busy with a single frame is 266 ms, and half of this (our average wait) is 133 ms. This is better, but still not perfect. The reason we choose this value (as compared to 64 or 128) is to provide good utilization of the line for bulk data transfers (such as large file transfers). Assuming a 5-byte CSLIP header, 256 bytes of data in a 261-byte frame gives 98.1% of the line to data and 1.9% to headers, which is good utilization. Reducing the MTU below 256 reduces the maximum throughput that we can achieve for bulk data transfers.

The MTU value listed in Figure 2.5, 296 for a point-to-point link, assumes 256 bytes of data and the 40-byte TCP and IP headers. Since the MTU is a value that IP queries the link layer for, the value must include the normal TCP and IP headers. This is how IP makes its fragmentation decision. IP knows nothing about the header compression that CSLIP performs.

Our average wait calculation (one-half the time required to transfer a maximum sized frame) only applies when a SLIP link (or PPP link) is used for both interactive traffic and bulk data transfer. When only interactive traffic is being exchanged, 1 byte of data in each direction (assuming 5-byte compressed headers) takes around 12.5 ms for the round trip at 9600 bits/sec. This is well within the 100–200 ms range mentioned earlier. Also notice that compressing the headers from 40 bytes to 5 bytes reduces the round-trip time for the 1 byte of data from 85 to 12.5 ms.

Unfortunately these types of calculations are harder to make when newer error correcting, compressing modems are being used. The compression employed by these modems reduces the number of bytes sent across the wire, but the error correction may increase the amount of time to transfer these bytes. Nevertheless, these calculations give us a starting point to make reasonable decisions.

In later chapters we'll use these serial line calculations to verify some of the timings that we see when watching packets go across a serial link.

2.11 Summary

This chapter has examined the lowest layer in the Internet protocol suite, the link layer. We looked at the difference between Ethernet and IEEE 802.2/802.3 encapsulation, and the encapsulation used by SLIP and PPP. Since both SLIP and PPP are often used on slow links, both provide a way to compress the common fields that don't often change. This provides better interactive response.

The loopback interface is provided by most implementations. Access to this interface is either through the special loopback address, normally 127.0.0.1, or by sending IP

3

IP: Internet Protocol

3.1 Introduction

IP is the workhorse protocol of the TCP/IP protocol suite. All TCP, UDP, ICMP, and IGMP data gets transmitted as IP datagrams (Figure 1.4). A fact that amazes many newcomers to TCP/IP, especially those from an X.25 or SNA background, is that IP provides an unreliable, connectionless datagram delivery service.

By *unreliable* we mean there are no guarantees that an IP datagram successfully gets to its destination. IP provides a best effort service. When something goes wrong, such as a router temporarily running out of buffers, IP has a simple error handling algorithm: throw away the datagram and try to send an ICMP message back to the source. Any required reliability must be provided by the upper layers (e.g., TCP).

The term *connectionless* means that IP does not maintain any state information about successive datagrams. Each datagram is handled independently from all other datagrams. This also means that IP datagrams can get delivered out of order. If a source sends two consecutive datagrams (first A, then B) to the same destination, each is routed independently and can take different routes, with B arriving before A.

In this chapter we take a brief look at the fields in the IP header, describe IP routing, and cover subnetting. We also look at two useful commands: `ifconfig` and `netstat`. We leave a detailed discussion of some of the fields in the IP header for later when we can see exactly how the fields are used. RFC 791 [Postel 1981a] is the official specification of IP.

3.2 IP Header

Figure 3.1 shows the format of an IP datagram. The normal size of the IP header is 20 bytes, unless options are present.

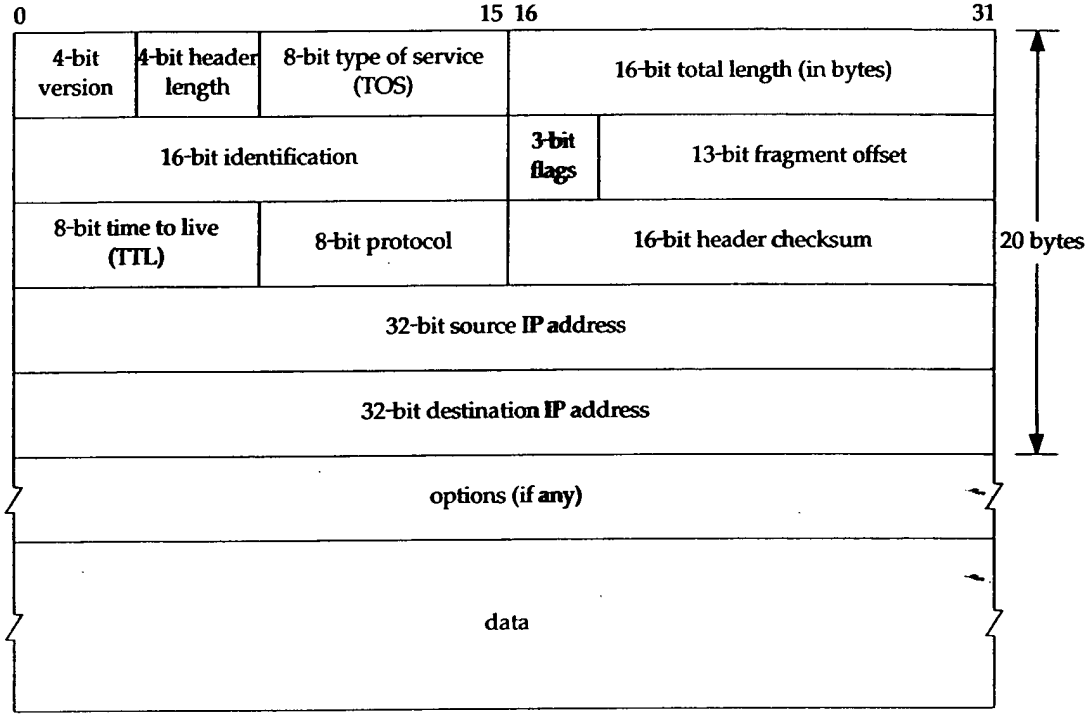


Figure 3.1 IP datagram, showing the fields in the IP header.

We will show the pictures of protocol headers in TCP/IP as in Figure 3.1. The most significant bit is numbered 0 at the left, and the least significant bit of a 32-bit value is numbered 31 on the right.

The 4 bytes in the 32-bit value are transmitted in the order: bits 0–7 first, then bits 8–15, then 16–23, and bits 24–31 last. This is called *big endian* byte ordering, which is the byte ordering required for all binary integers in the TCP/IP headers as they traverse a network. This is called the *network byte order*. Machines that store binary integers in other formats, such as the *little endian* format, must convert the header values into the network byte order before transmitting the data.

The current protocol *version* is 4, so IP is sometimes called IPv4. Section 3.10 discusses some proposals for a new version of IP.

The *header length* is the number of 32-bit words in the header, including any options. Since this is a 4-bit field, it limits the header to 60 bytes. In Chapter 8 we'll see that this limitation makes some of the options, such as the record route option, useless today. The normal value of this field (when no options are present) is 5.

The *type-of-service* field (TOS) is composed of a 3-bit precedence field (which is ignored today), 4 TOS bits, and an unused bit that must be 0. The 4 TOS bits are: minimize delay, maximize throughput, maximize reliability, and minimize monetary cost.

Only 1 of these 4 bits can be turned on. If all 4 bits are 0 it implies normal service. RFC 1340 [Reynolds and Postel 1992] specifies how these bits should be set by all the standard applications. RFC 1349 [Almquist 1992] contains some corrections to this RFC, and a more detailed description of the TOS feature.

Figure 3.2 shows the recommended values of the TOS field for various applications. In the final column we show the hexadecimal value, since that's what we'll see in the `tcpdump` output later in the text.

Application	Minimize delay	Maximize throughput	Maximize reliability	Minimize monetary cost	Hex value
Telnet/Rlogin	1	0	0	0	0x10
FTP					
control	1	0	0	0	0x10
data	0	1	0	0	0x08
any bulk data	0	1	0	0	0x08
TFTP	1	0	0	0	0x10
SMTP					
command phase	1	0	0	0	0x10
data phase	0	1	0	0	0x08
DNS					
UDP query	1	0	0	0	0x10
TCP query	0	0	0	0	0x00
zone transfer	0	1	0	0	0x08
ICMP					
error	0	0	0	0	0x00
query	0	0	0	0	0x00
any IGP	0	0	1	0	0x04
SNMP	0	0	1	0	0x04
BOOTP	0	0	0	0	0x00
NNTP	0	0	0	1	0x02

Figure 3.2 Recommended values for type-of-service field.

The interactive login applications, Telnet and Rlogin, want a minimum delay since they're used interactively by a human for small amounts of data transfer. File transfer by FTP, on the other hand, wants maximum throughput. Maximum reliability is specified for network management (SNMP) and the routing protocols. Usenet news (NNTP) is the only one shown that wants to minimize monetary cost.

The TOS feature is not supported by most TCP/IP implementations today, though newer systems starting with 4.3BSD Reno are setting it. Additionally, new routing protocols such as OSPF and IS–IS are capable of making routing decisions based on this field.

> In Section 2.10 we mentioned that SLIP drivers normally provide type-of-service queueing, allowing interactive traffic to be handled before bulk data. Since most implementations don't use the TOS field, this queueing is done ad hoc by SLIP, with the driver looking at the protocol field (to determine whether it's a TCP segment or not) and then checking the source and destination TCP port numbers to see if the port number corresponds to an interactive service. One driver comments that this "disgusting hack" is required since most implementations don't allow the application to set the TOS field.

The *total length* field is the total length of the IP datagram in bytes. Using this field and the header length field, we know where the data portion of the IP datagram starts, and its length. Since this is a 16-bit field, the maximum size of an IP datagram is 65535 bytes. (Recall from Figure 2.5 [p. 30] that a Hyperchannel has an MTU of 65535. This means there really isn't an MTU—it uses the largest IP datagram possible.) This field also changes when a datagram is fragmented, which we describe in Section 11.5.

Although it's possible to send a 65535-byte IP datagram, most link layers will fragment this. Furthermore, a host is not required to receive a datagram larger than 576 bytes. TCP divides the user's data into pieces, so this limit normally doesn't affect TCP. With UDP we'll encounter numerous applications in later chapters (RIP, TFTP, BOOTP, the DNS, and SNMP) that limit themselves to 512 bytes of user data, to stay below this 576-byte limit. Realistically, however, most implementations today (especially those that support the Network File System, NFS) allow for just over 8192-byte IP datagrams.

The total length field is required in the IP header since some data links (e.g., Ethernet) pad small frames to be a minimum length. Even though the minimum Ethernet frame size is 46 bytes (Figure 2.1), an IP datagram can be smaller. If the total length field wasn't provided, the IP layer wouldn't know how much of a 46-byte Ethernet frame was really an IP datagram.

The *identification* field uniquely identifies each datagram sent by a host. It normally increments by one each time a datagram is sent. We return to this field when we look at fragmentation and reassembly in Section 11.5. Similarly, we'll also look at the *flags* field and the *fragmentation offset* field when we talk about fragmentation.

> RFC 791 [Postel 1981a] says that the identification field should be chosen by the upper layer that is having IP send the datagram. This implies that two consecutive IP datagrams, one generated by TCP and one generated by UDP, can have the same identification field. While this is OK (the reassembly algorithm handles this), most Berkeley-derived implementations have the IP layer increment a kernel variable each time an IP datagram is sent, regardless of which layer passed the data to IP to send. This kernel variable is initialized to a value based on the time-of-day when the system is bootstrapped.

The *time-to-live* field, or *TTL*, sets an upper limit on the number of routers through which a datagram can pass. It limits the lifetime of the datagram. It is initialized by the sender to some value (often 32 or 64) and decremented by one by every router that handles the datagram. When this field reaches 0, the datagram is thrown away, and the sender is notified with an ICMP message. This prevents packets from getting caught in routing loops forever. We return to this field in Chapter 8 when we look at the Traceroute program.

We talked about the *protocol* field in Chapter 1 and showed how it is used by IP to demultiplex incoming datagrams in Figure 1.8. It identifies which protocol gave the data for IP to send.

The *header checksum* is calculated over the IP header only. It does *not* cover any data that follows the header. ICMP, IGMP, UDP, and TCP all have a checksum in their own headers to cover their header and data.

To compute the IP checksum for an outgoing datagram, the value of the checksum field is first set to 0. Then the 16-bit one's complement sum of the header is calculated (i.e., the entire header is considered a sequence of 16-bit words). The 16-bit one's

complement of this sum is stored in the checksum field. When an IP datagram is received, the 16-bit one's complement sum of the header is calculated. Since the receiver's calculated checksum contains the checksum stored by the sender, the receiver's checksum is all one bits if nothing in the header was modified. If the result is not all one bits (a checksum error), IP discards the received datagram. No error message is generated. It is up to the higher layers to somehow detect the missing datagram and retransmit.

ICMP, IGMP, UDP, and TCP all use the same checksum algorithm, although TCP and UDP include various fields from the IP header, in addition to their own header and data. RFC 1071 [Braden, Borman, and Partridge 1988] contains implementation techniques for computing the Internet checksum. Since a router often changes only the TTL field (decrementing it by 1), a router can incrementally update the checksum when it forwards a received datagram, instead of calculating the checksum over the entire IP header again. RFC 1141 [Mallory and Kullberg 1990] describes an efficient way to do this.

> The standard BSD implementation, however, does not use this incremental update feature when forwarding a datagram.

Every IP datagram contains the *source IP address* and the *destination IP address*. These are the 32-bit values that we described in Section 1.4.

The final field, the *options*, is a variable-length list of optional information for the datagram. The options currently defined are:

- security and handling restrictions (for military applications, refer to RFC 1108 [Kent 1991] for details),
- record route (have each router record its IP address, Section 7.3),
- timestamp (have each router record its IP address and time, Section 7.4),
- loose source routing (specifying a list of IP addresses that must be traversed by the datagram, Section 8.5), and
- strict source routing (similar to loose source routing but here only the addresses in the list can be traversed, Section 8.5).

These options are rarely used and not all host and routers support all the options.

The options field always ends on a 32-bit boundary. Pad bytes with a value of 0 are added if necessary. This assures that the IP header is always a multiple of 32 bits (as required for the *header length* field).

3.3 IP Routing

Conceptually, IP routing is simple, especially for a host. If the destination is directly connected to the host (e.g., a point-to-point link) or on a shared network (e.g., Ethernet or token ring), then the IP datagram is sent directly to the destination. Otherwise the

host sends the datagram to a default router, and lets the router deliver the datagram to its destination. This simple scheme handles most host configurations.

In this section and in Chapter 9 we'll look at the more general case where the IP layer can be configured to act as a router in addition to acting as a host. Most multiuser systems today, including almost every Unix system, can be configured to act as a router. We can then specify a single routing algorithm that both hosts and routers can use. The fundamental difference is that a host *never* forwards datagrams from one of its interfaces to another, while a router forwards datagrams. A host that contains embedded router functionality should never forward a datagram unless it has been specifically configured to do so. We say more about this configuration option in Section 9.4.

In our general scheme, IP can receive a datagram from TCP, UDP, ICMP, or IGMP (that is, a locally generated datagram) to send, or one that has been received from a network interface (a datagram to forward). The IP layer has a routing table in memory that it searches each time it receives a datagram to send. When a datagram is received from a network interface, IP first checks if the destination IP address is one of its own IP addresses or an IP broadcast address. If so, the datagram is delivered to the protocol module specified by the protocol field in the IP header. If the datagram is not destined for this IP layer, then (1) if the IP layer was configured to act as a router the packet is forwarded (that is, handled as an outgoing datagram as described below), else (2) the datagram is silently discarded.

Each entry in the routing table contains the following information:

- Destination IP address. This can be either a complete *host address* or a *network address*, as specified by the flag field (described below) for this entry. A host address has a nonzero host ID (Figure 1.5) and identifies one particular host, while a network address has a host ID of 0 and identifies all the hosts on that network (e.g., Ethernet, token ring).
- IP address of a *next-hop router*, or the IP address of a directly connected network. A next-hop router is one that is on a directly connected network to which we can send datagrams for delivery. The next-hop router is not the final destination, but it takes the datagrams we send it and forwards them to the final destination.
- Flags. One flag specifies whether the destination IP address is the address of a network or the address of a host. Another flag says whether the next-hop router field is really a next-hop router or a directly connected interface. (We describe each of these flags in Section 9.2.)
- Specification of which network interface the datagram should be passed to for transmission.

IP routing is done on a hop-by-hop basis. As we can see from this routing table information, IP does not know the complete route to any destination (except, of course, those destinations that are directly connected to the sending host). All that IP routing provides is the IP address of the next-hop router to which the datagram is sent. It is assumed that the next-hop router is really "closer" to the destination than the sending host is, and that the next-hop router is directly connected to the sending host.

IP routing performs the following actions:

1. Search the routing table for an entry that matches the complete destination IP address (matching network ID and host ID). If found, send the packet to the indicated next-hop router or to the directly connected interface (depending on the flags field). Point-to-point links are found here, for example, since the other end of such a link is the other host's complete IP address.
2. Search the routing table for an entry that matches just the destination network ID. If found, send the packet to the indicated next-hop router or to the directly connected interface (depending on the flags field). All the hosts on the destination network can be handled with this single routing table entry. All the hosts on a local Ethernet, for example, are handled with a routing table entry of this type.

   This check for a network match must take into account a possible subnet mask, which we describe in the next section.
3. Search the routing table for an entry labeled "default." If found, send the packet to the indicated next-hop router.

If none of the steps works, the datagram is undeliverable. If the undeliverable datagram was generated on this host, a "host unreachable" or "network unreachable" error is normally returned to the application that generated the datagram.

A complete matching host address is searched for before a matching network ID. Only if both of these fail is a default route used. Default routes, along with the ICMP redirect message sent by a next-hop router (if we chose the wrong default for a datagram), are powerful features of IP routing that we'll come back to in Chapter 9.

The ability to specify a route to a network, and not have to specify a route to every host, is another fundamental feature of IP routing. Doing this allows the routers on the Internet, for example, to have a routing table with thousands of entries, instead of a routing table with more than one million entries.

Examples

First consider a simple example: our host `bsdi` has an IP datagram to send to our host `sun`. Both hosts are on the same Ethernet (see inside front cover). Figure 3.3 shows the delivery of the datagram.

When IP receives the datagram from one of the upper layers it searches its routing table and finds that the destination IP address (140.252.13.33) is on a directly connected network (the Ethernet 140.252.13.0). A matching network address is found in the routing table. (In the next section we'll see that because of subnetting the network address of this Ethernet is really 140.252.13.32, but that doesn't affect this discussion of routing.)

The datagram is passed to the Ethernet device driver, and sent to `sun` as an Ethernet frame (Figure 2.1). The destination address in the IP datagram is Sun's IP address (140.252.13.33) and the destination address in the link-layer header is the 48-bit Ethernet address of `sun`'s Ethernet interface. This 48-bit Ethernet address is obtained using ARP, as we describe in the next chapter.

17

TCP: Transmission Control Protocol

17.1 Introduction

In this chapter we provide a description of the services provided by TCP for the application layer. We also look at the fields in the TCP header. In the chapters that follow we examine all of these header fields in more detail, as we see how TCP operates.

Our description of TCP starts in this chapter and continues in the next seven chapters. Chapter 18 describes how a TCP connection is established and terminated, and Chapters 19 and 20 look at the normal transfer of data, both for interactive use (remote login) and bulk data (file transfer). Chapter 21 provides the details of TCP's timeout and retransmission, followed by two other TCP timers in Chapters 22 and 23. Finally Chapter 24 takes a look at newer TCP features and TCP performance.

The original specification for TCP is RFC 793 [Postel 1981c], although some errors in that RFC are corrected in the Host Requirements RFC.

17.2 TCP Services

Even though TCP and UDP use the same network layer (IP), TCP provides a totally different service to the application layer than UDP does. TCP provides a connection-oriented, reliable, byte stream service.

The term *connection-oriented* means the two applications using TCP (normally considered a client and a server) must establish a TCP connection with each other before they can exchange data. The typical analogy is dialing a telephone number, waiting for the other party to answer the phone and say "hello," and then saying who's calling. In Chapter 18 we look at how a connection is established, and disconnected some time later when either end is done.

There are exactly two end points communicating with each other on a TCP connection. Concepts that we talked about in Chapter 12, broadcasting and multicasting, aren't applicable to TCP.

TCP provides *reliability* by doing the following:

- The application data is broken into what TCP considers the best sized chunks to send. This is totally different from UDP, where each write by the application generates a UDP datagram of that size. The unit of information passed by TCP to IP is called a *segment*. (See Figure 1.7, p. 10.) In Section 18.4 we'll see how TCP decides what this segment size is.
- When TCP sends a segment it maintains a timer, waiting for the other end to acknowledge reception of the segment. If an acknowledgment isn't received in time, the segment is retransmitted. In Chapter 21 we'll look at TCP's adaptive timeout and retransmission strategy.
- When TCP receives data from the other end of the connection, it sends an acknowledgment. This acknowledgment is not sent immediately, but normally delayed a fraction of a second, as we discuss in Section 19.3.
- TCP maintains a checksum on its header and data. This is an end-to-end checksum whose purpose is to detect any modification of the data in transit. If a segment arrives with an invalid checksum, TCP discards it and doesn't acknowledge receiving it. (It expects the sender to time out and retransmit.)
- Since TCP segments are transmitted as IP datagrams, and since IP datagrams can arrive out of order, TCP segments can arrive out of order. A receiving TCP resequences the data if necessary, passing the received data in the correct order to the application.
- Since IP datagrams can get duplicated, a receiving TCP must discard duplicate data.
- TCP also provides flow control. Each end of a TCP connection has a finite amount of buffer space. A receiving TCP only allows the other end to send as much data as the receiver has buffers for. This prevents a fast host from taking all the buffers on a slower host.

A stream of 8-bit bytes is exchanged across the TCP connection between the two applications. There are no record markers automatically inserted by TCP. This is what we called a *byte stream service*. If the application on one end writes 10 bytes, followed by a write of 20 bytes, followed by a write of 50 bytes, the application at the other end of the connection cannot tell what size the individual writes were. The other end may read the 80 bytes in four reads of 20 bytes at a time. One end puts a stream of bytes into TCP and the same, identical stream of bytes appears at the other end.

Also, TCP does not interpret the contents of the bytes at all. TCP has no idea if the data bytes being exchanged are binary data, ASCII characters, EBCDIC characters, or whatever. The interpretation of this byte stream is up to the applications on each end of the connection.

This treatment of the byte stream by TCP is similar to the treatment of a file by the Unix operating system. The Unix kernel does no interpretation whatsoever of the bytes that an application reads or write—that is up to the applications. There is no distinction to the Unix kernel between a binary file or a file containing lines of text.

17.3 TCP Header

Recall that TCP data is encapsulated in an IP datagram, as shown in Figure 17.1.

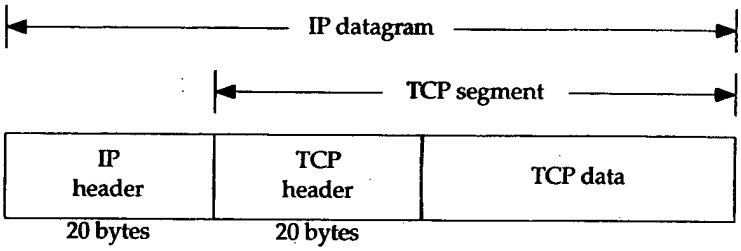


Figure 17.1 Encapsulation of TCP data in an IP datagram.

Figure 17.2 shows the format of the TCP header. Its normal size is 20 bytes, unless options are present.

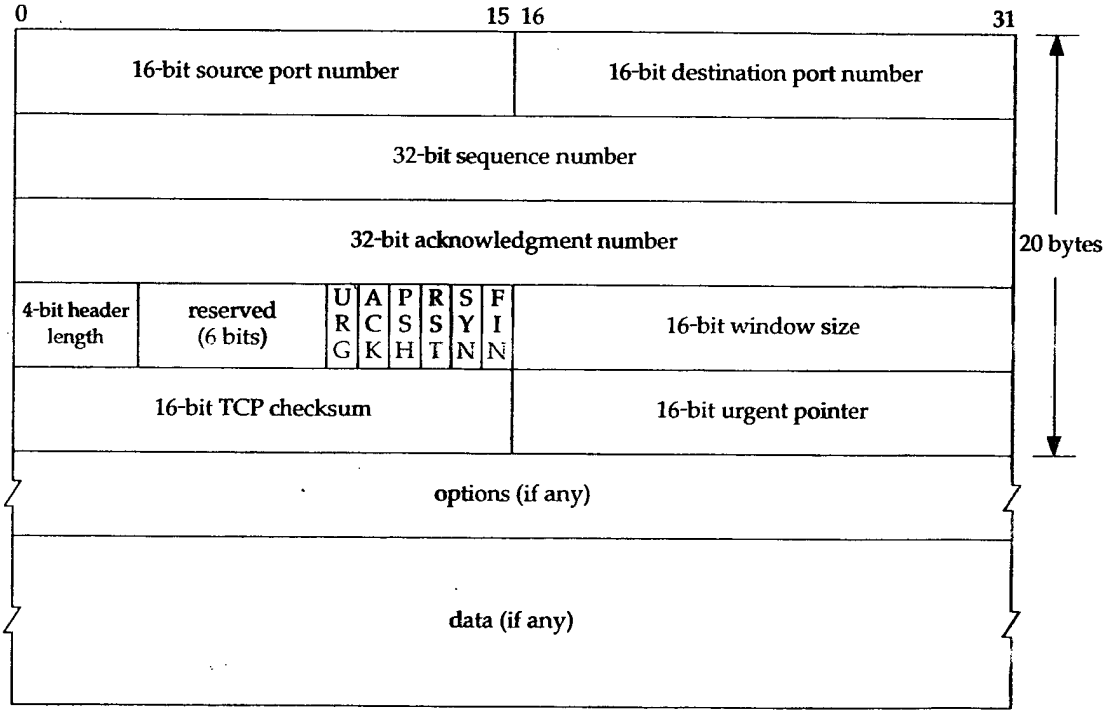


Figure 17.2 TCP header.

Each TCP segment contains the source and destination *port number* to identify the sending and receiving application. These two values, along with the source and destination IP addresses in the IP header, uniquely identify each *connection*.

The combination of an IP address and a port number is sometimes called a *socket*. This term appeared in the original TCP specification (RFC 793), and later it also became used as the name of the Berkeley-derived programming interface (Section 1.15). It is the *socket pair* (the 4-tuple consisting of the client IP address, client port number, server IP address, and server port number) that specifies the two end points that uniquely identifies each TCP connection in an internet.

The *sequence number* identifies the byte in the stream of data from the sending TCP to the receiving TCP that the first byte of data in this segment represents. If we consider the stream of bytes flowing in one direction between two applications, TCP numbers each byte with a sequence number. This sequence number is a 32-bit unsigned number that wraps back around to 0 after reaching $2^{32} - 1$.

When a new connection is being established, the SYN flag is turned on. The *sequence number field* contains the *initial sequence number* (ISN) chosen by this host for this connection. The sequence number of the first byte of data sent by this host will be the ISN plus one because the SYN flag consumes a sequence number. (We describe additional details on exactly how a connection is established and terminated in the next chapter where we'll see that the FIN flag consumes a sequence number also.)

Since every byte that is exchanged is numbered, the *acknowledgment number* contains the next sequence number that the sender of the acknowledgment expects to receive. This is therefore the sequence number plus 1 of the last successfully received byte of data. This field is valid only if the ACK flag (described below) is on.

Sending an ACK costs nothing because the 32-bit acknowledgment number field is always part of the header, as is the ACK flag. Therefore we'll see that once a connection is established, this field is always set and the ACK flag is always on.

TCP provides a *full-duplex* service to the application layer. This means that data can be flowing in each direction, independent of the other direction. Therefore, each end of a connection must maintain a sequence number of the data flowing in each direction.

TCP can be described as a sliding-window protocol without selective or negative acknowledgments. (The sliding window protocol used for data transmission is described in Section 20.3.) We say that TCP lacks selective acknowledgments because the acknowledgment number in the TCP header means that the sender has successfully received up through but not including that byte. There is currently no way to acknowledge selected pieces of the data stream. For example, if bytes 1–1024 are received OK, and the next segment contains bytes 2049–3072, the receiver cannot acknowledge this new segment. All it can send is an ACK with 1025 as the acknowledgment number. There is no means for negatively acknowledging a segment. For example, if the segment with bytes 1025–2048 did arrive, but had a checksum error, all the receiving TCP can send is an ACK with 1025 as the acknowledgment number. In Section 21.7 we'll see how duplicate acknowledgments can help determine that packets have been lost.

The *header length* gives the length of the header in 32-bit words. This is required because the length of the options field is variable. With a 4-bit field, TCP is limited to a 60-byte header. Without options, however, the normal size is 20 bytes.

There are six flag bits in the TCP header. One or more of them can be turned on at the same time. We briefly mention their use here and discuss each flag in more detail in later chapters.

URG The *urgent pointer* is valid (Section 20.8).

ACK The *acknowledgment number* is valid.

PSH The receiver should pass this data to the application as soon as possible (Section 20.5).

RST Reset the connection (Section 18.7).

SYN Synchronize sequence numbers to initiate a connection. This flag and the next are described in Chapter 18.

FIN The sender is finished sending data.

TCP's flow control is provided by each end advertising a *window size*. This is the number of bytes, starting with the one specified by the acknowledgment number field, that the receiver is willing to accept. This is a 16-bit field, limiting the window to 65535 bytes. In Section 24.4 we'll look at the new window scale option that allows this value to be scaled, providing larger windows.

The *checksum* covers the TCP segment: the TCP header and the TCP data. This is a mandatory field that must be calculated and stored by the sender, and then verified by the receiver. The TCP checksum is calculated similar to the UDP checksum, using a pseudo-header as described in Section 11.3.

The *urgent pointer* is valid only if the URG flag is set. This pointer is a positive offset that must be added to the sequence number field of the segment to yield the sequence number of the last byte of urgent data. TCP's urgent mode is a way for the sender to transmit emergency data to the other end. We'll look at this feature in Section 20.8.

The most common *option* field is the maximum segment size option, called the *MSS*. Each end of a connection normally specifies this option on the first segment exchanged (the one with the SYN flag set to establish the connection). It specifies the maximum sized segment that the sender wants to receive. We describe the MSS option in more detail in Section 18.4, and some of the other TCP options in Chapter 24.

In Figure 17.2 we note that the data portion of the TCP segment is optional. We'll see in Chapter 18 that when a connection is established, and when a connection is terminated, segments are exchanged that contain only the TCP header with possible options. A header without any data is also used to acknowledge received data, if there is no data to be transmitted in that direction. There are also some cases dealing with timeouts when a segment can be sent without any data.

17.4 Summary

TCP provides a reliable, connection-oriented, byte stream, transport layer service. We looked briefly at all the fields in the TCP header and will examine them in detail in the following chapters.

TCP packetizes the user data into segments, sets a timeout any time it sends data, acknowledges data received by the other end, reorders out-of-order data, discards duplicate data, provides end-to-end flow control, and calculates and verifies a mandatory end-to-end checksum.

TCP is used by many of the popular applications, such as Telnet, Rlogin, FTP, and electronic mail (SMTP).

Exercises

17.1 We've covered the following packet formats, each of which has a checksum in its corresponding header: IP, ICMP, IGMP, UDP, and TCP. For each one, describe what portion of an IP datagram the checksum covers and whether the checksum is mandatory or optional.

17.2 Why do all the Internet protocols that we've discussed (IP, ICMP, IGMP, UDP, TCP) quietly discard a packet that arrives with a checksum error?

17.3 TCP provides a byte-stream service where record boundaries are not maintained between the sender and receiver. How can applications provide their own record markers?

17.4 Why are the source and destination port numbers at the beginning of the TCP header?

17.5 Why does the TCP header have a header length field while the UDP header (Figure 11.2, p. 144) does not?

18

TCP Connection Establishment and Termination

18.1 Introduction

TCP is a *connection-oriented* protocol. Before either end can send data to the other, a *connection* must be established between them. In this chapter we take a detailed look at how a TCP connection is established and later terminated.

This establishment of a connection between the two ends differs from a *connectionless* protocol such as UDP. We saw in Chapter 11 that with UDP one end just sends a datagram to the other end, without any preliminary handshaking.

18.2 Connection Establishment and Termination

To see what happens when a TCP connection is established and then terminated, we type the following command on the system `svr4`:

```
svr4 % telnet bsdi discard
Trying 140.252.13.35 ...
Connected to bsdi.
Escape character is '^]'.
^]                                  type Control, right bracket to talk to the Telnet client
telnet> quit                        terminate the connection
Connection closed.
```

The `telnet` command establishes a TCP connection with the host `bsdi` on the port corresponding to the discard service (Section 1.12). This is exactly the type of service we need to see what happens when a connection is established and terminated, without having the server initiate any data exchange.

tcpdump Output

Figure 18.1 shows the tcpdump output for the segments generated by this command.

```
1  0.0                    svr4.1037 > bsdi.discard: S 1415531521:1415531521(0)
                                                     win 4096 <mss 1024>
2  0.002402 (0.0024)      bsdi.discard > svr4.1037: S 1823083521:1823083521(0)
                                                     ack 1415531522 win 4096
                                                     <mss 1024>
3  0.007224 (0.0048)      svr4.1037 > bsdi.discard: . ack 1823083522 win 4096
4  4.155441 (4.1482)      svr4.1037 > bsdi.discard: F 1415531522:1415531522(0)
                                                     ack 1823083522 win 4096
5  4.156747 (0.0013)      bsdi.discard > svr4.1037: . ack 1415531523 win 4096
6  4.158144 (0.0014)      bsdi.discard > svr4.1037: F 1823083522:1823083522(0)
                                                     ack 1415531523 win 4096
7  4.180662 (0.0225)      svr4.1037 > bsdi.discard: . ack 1823083523 win 4096
```

Figure 18.1 tcpdump output for TCP connection establishment and termination.

These seven TCP segments contain TCP headers only. No data is exchanged.

For TCP segments, each output line begins with

source > *destination: flags*

where *flags* represents four of the six flag bits in the TCP header (Figure 17.2). Figure 18.2 shows the five different characters that can appear in the *flags* output.

flag	3-character abbreviation	Description
S	SYN	synchronize sequence numbers
F	FIN	sender is finished sending data
R	RST	reset connection
P	PSH	push data to receiving process as soon as possible
.		none of above four flags is on

Figure 18.2 *flag* characters output by tcpdump for flag bits in TCP header.

In this example we see the S, F, and period. We'll see the other two *flags* (R and P) later. The other two TCP header flag bits—ACK and URG—are printed specially by tcpdump.

It's possible for more than one of the four flag bits in Figure 18.2 to be on in a single segment, but we normally see only one on at a time.

> RFC 1025 [Postel 1987], the *TCP and IP Bake Off*, calls a segment with the maximum combination of allowable flag bits turned on at once (SYN, URG, PSH, FIN, and 1 byte of data) a Kamikaze packet. It's also known as a nastygram, Christmas tree packet, and lamp test segment.

In line 1, the field `1415531521:1415531521(0)` means the sequence number of the packet was 1415531521 and the number of data bytes in the segment was 0. `tcpdump` displays this by printing the starting sequence number, a colon, the implied ending sequence number, and the number of data bytes in parentheses. The advantage of displaying both the sequence number and the implied ending sequence number is to see what the implied ending sequence number is, when the number of bytes is greater than 0. This field is output only if (1) the segment contains one or more bytes of data or (2) the SYN, FIN, or RST flag was on. Lines 1, 2, 4, and 6 in Figure 18.1 display this field because of the flag bits—we never exchange any data in this example.

In line 2 the field `ack 1415531522` shows the acknowledgment number. This is printed only if the ACK flag in the header is on.

The field `win 4096` in every line of output shows the window size being advertised by the sender. In these examples, where we are not exchanging any data, the window size never changes from its default of 4096. (We examine TCP's window size in Section 20.4.)

The final field that is output in Figure 18.1, `<mss 1024>` shows the *maximum segment size* (MSS) option specified by the sender. The sender does not want to receive TCP segments larger than this value. This is normally to avoid fragmentation (Section 11.5). We discuss the maximum segment size in Section 18.4, and show the format of the various TCP options in Section 18.10.

Time Line

Figure 18.3 shows the time line for this sequence of packets. (We described some general features of these time lines when we showed the first one in Figure 6.11, p. 80.) This figure shows which end is sending packets. We also expand some of the `tcpdump` output (e.g., printing SYN instead of `S`). In this time line we have also removed the window size values, since they add nothing to the discussion.

Connection Establishment Protocol

Now let's return to the details of the TCP protocol that are shown in Figure 18.3. To establish a TCP connection:

1. The requesting end (normally called the *client*) sends a SYN segment specifying the port number of the *server* that the client wants to connect to, and the client's *initial sequence number* (ISN, 1415531521 in this example). This is segment 1.
2. The server responds with its own SYN segment containing the server's initial sequence number (segment 2). The server also acknowledges the client's SYN by ACKing the client's ISN plus one. A SYN consumes one sequence number.
3. The client must acknowledge this SYN from the server by ACKing the server's ISN plus one (segment 3).

These three segments complete the connection establishment. This is often called the *three-way handshake*.

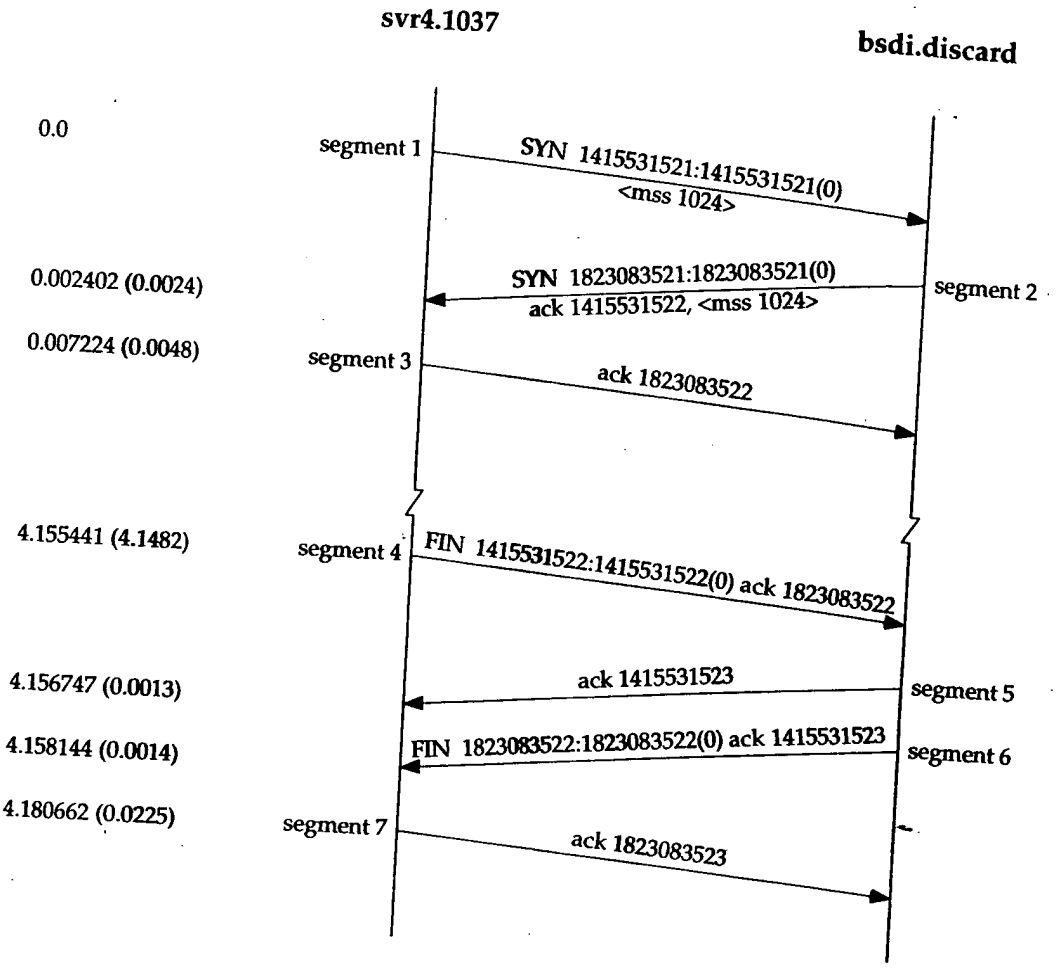


Figure 18.3 Time line of connection establishment and connection termination.

The side that sends the first SYN is said to perform an *active open*. The other side, which receives this SYN and sends the next SYN, performs a *passive open*. (In Section 18.8 we describe a simultaneous open where both sides can do an active open.)

When each end sends its SYN to establish the connection, it chooses an initial sequence number for that connection. The ISN should change over time, so that each connection has a different ISN. RFC 793 [Postel 1981c] specifies that the ISN should be viewed as a 32-bit counter that increments by one every 4 microseconds. The purpose in these sequence numbers is to prevent packets that get delayed in the network from being delivered later and then misinterpreted as part of an existing connection.

> How are the sequence numbers chosen? In 4.4BSD (and most Berkeley-derived implementations) when the system is initialized the initial send sequence number is initialized to 1. This practice violates the Host Requirements RFC. (A comment in the code acknowledges that this is wrong.) This variable is then incremented by 64,000 every half-second, and will cycle back to 0 about every 9.5 hours. (This corresponds to a counter that is incremented every 8

microseconds, not every 4 microseconds.) Additionally, each time a connection is established, this variable is incremented by 64,000.

The 4.1-second gap between segments 3 and 4 is the time between establishing the connection and typing the `quit` command to `telnet` to terminate the connection.

Connection Termination Protocol

While it takes three segments to establish a connection, it takes four to terminate a connection. This is caused by TCP's *half-close*. Since a TCP connection is full-duplex (that is, data can be flowing in each direction independently of the other direction), each direction must be shut down independently. The rule is that either end can send a FIN when it is done sending data. When a TCP receives a FIN, it must notify the application that the other end has terminated that direction of data flow. The sending of a FIN is normally the result of the application issuing a close.

The receipt of a FIN only means there will be no more data flowing in that direction. A TCP can still send data after receiving a FIN. While it's possible for an application to take advantage of this half-close, in practice few TCP applications use it. The normal scenario is what we show in Figure 18.3. We describe the half-close in more detail in Section 18.5.

We say that the end that first issues the close (e.g., sends the first FIN) performs the *active close* and the other end (that receives this FIN) performs the *passive close*. Normally one end does the active close and the other does the passive close, but we'll see in Section 18.9 how both ends can do an active close.

Segment 4 in Figure 18.3 initiates the termination of the connection and is sent when the Telnet client closes its connection. This happens when we type `quit`. This causes the client TCP to send a FIN, closing the flow of data from the client to the server.

When the server receives the FIN it sends back an ACK of the received sequence number plus one (segment 5). A FIN consumes a sequence number, just like a SYN. At this point the server's TCP also delivers an end-of-file to the application (the discard server). The server then closes its connection, causing its TCP to send a FIN (segment 6), which the client TCP must ACK by incrementing the received sequence number by one (segment 7).

Figure 18.4 shows the typical sequence of segments that we've described for the termination of a connection. We omit the sequence numbers. In this figure sending the FINs is caused by the applications closing their end of the connection, whereas the ACKs of these FINs are automatically generated by the TCP software.

Connections are normally initiated by the client, with the first SYN going from the client to the server. Either end can actively close the connection (i.e., send the first FIN). Often, however, it is the client that determines when the connection should be terminated, since client processes are often driven by an interactive user, who enters something like "quit" to terminate. In Figure 18.4 we can switch the labels at the top, calling the left side the server and the right side the client, and everything still works fine as shown. (The first example in Section 14.4, for example, shows the daytime server closing the connection.)

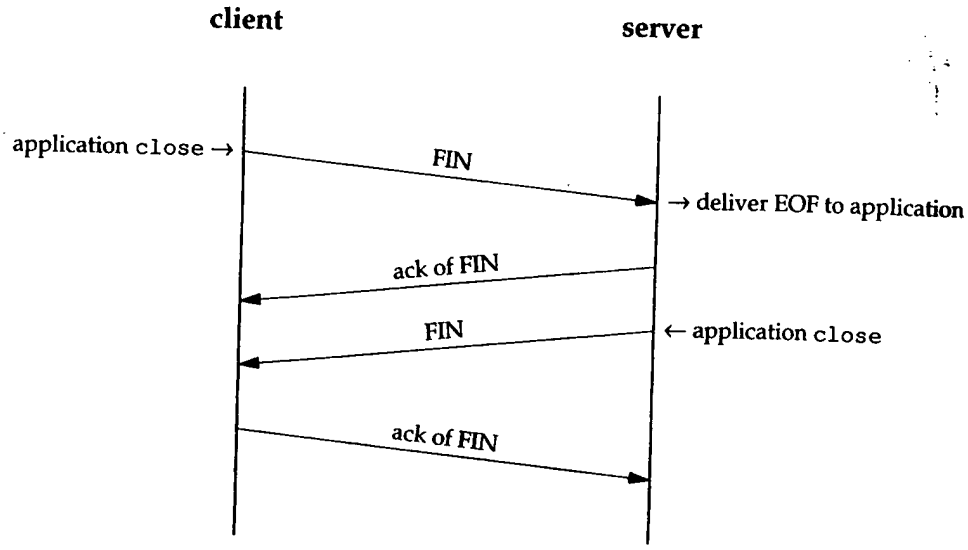


Figure 18.4 Normal exchange of segments during connection termination.

Normal `tcpdump` Output

Having to sort through all the huge sequence numbers is cumbersome, so the default `tcpdump` output shows the complete sequence numbers only on the SYN segments, and shows all following sequence numbers as relative offsets from the original sequence numbers. (To generate the output for Figure 18.1 we had to specify the `-S` option.) The normal `tcpdump` output corresponding to Figure 18.1 is shown in Figure 18.5.

```
1  0.0                    svr4.1037 > bsdi.discard: S 1415531521:1415531521(0)
                                                    win 4096 <mss 1024>
2  0.002402 (0.0024)      bsdi.discard > svr4.1037: S 1823083521:1823083521(0)
                                                    ack 1415531522
                                                    win 4096 <mss 1024>
3  0.007224 (0.0048)      svr4.1037 > bsdi.discard: . ack 1 win 4096
4  4.155441 (4.1482)      svr4.1037 > bsdi.discard: F 1:1(0) ack 1 win 4096
5  4.156747 (0.0013)      bsdi.discard > svr4.1037: . ack 2 win 4096
6  4.158144 (0.0014)      bsdi.discard > svr4.1037: F 1:1(0) ack 2 win 4096
7  4.180662 (0.0225)      svr4.1037 > bsdi.discard: . ack 2 win 4096
```

Figure 18.5 Normal `tcpdump` output for connection establishment and termination.

Unless we need to show the complete sequence numbers, we'll use this form of output in all following examples.

18.3 Timeout of Connection Establishment

There are several instances when the connection cannot be established. In one example the server host is down. To simulate this scenario we issue our `telnet` command after disconnecting the Ethernet cable from the server's host. Figure 18.6 shows the `tcpdump` output.

```
1   0.0                  bsdi.1024 > svr4.discard: S 291008001:291008001(0)
                                                    win 4096 <mss 1024>
                                                    [tos 0x10]
2   5.814797 ( 5.8148)   bsdi.1024 > svr4.discard: S 291008001:291008001(0)
                                                    win 4096 <mss 1024>
                                                    [tos 0x10]
3  29.815436 (24.0006)   bsdi.1024 > svr4.discard: S 291008001:291008001(0)
                                                    win 4096 <mss 1024>
                                                    [tos 0x10]
```

Figure 18.6 `tcpdump` output for connection establishment that times out.

The interesting point in this output is how frequently the client's TCP sends a SYN to try to establish the connection. The second segment is sent 5.8 seconds after the first, and the third is sent 24 seconds after the second.

> As a side note, this example was run about 38 minutes after the client was rebooted. This corresponds with the initial sequence number of 291,008,001 (approximately $38 \times 60 \times 64000 \times 2$). Recall earlier in this chapter we said that typical Berkeley-derived systems initialize the initial sequence number to 1 and then increment it by 64,000 every half-second.
>
> Also, this is the first TCP connection since the system was bootstrapped, which is why the client's port number is 1024.

What isn't shown in Figure 18.6 is how long the client's TCP keeps retransmitting before giving up. To see this we have to time the `telnet` command:

```
bsdi % date ; telnet svr4 discard ; date
Thu Sep 24 16:24:11 MST 1992
Trying 140.252.13.34...
telnet: Unable to connect to remote host: Connection timed out
Thu Sep 24 16:25:27 MST 1992
```

The time difference is 76 seconds. Most Berkeley-derived systems set a time limit of 75 seconds on the establishment of a new connection. We'll see in Section 21.4 that the third packet sent by the client would have timed out around 16:25:29, 48 seconds after it was sent, had the client not given up after 75 seconds.

First Timeout Period

One puzzling item in Figure 18.6 is that the first timeout period, 5.8 seconds, is close to 6 seconds, but not exact, while the second period is almost exactly 24 seconds. Ten more

of these tests were run and the first timeout period took on various values between 5.59 seconds and 5.93 seconds. The second timeout period, however, was always 24.00 (to two decimal places).

What's happening here is that BSD implementations of TCP run a timer that goes off every 500 ms. This 500-ms timer is used for various TCP timeouts, all of which we cover in later chapters. When we type in the `telnet` command, an initial 6-second timer is established (12 clock ticks), but it may expire anywhere between 5.5 and 6 seconds in the future. Figure 18.7 shows what's happening.

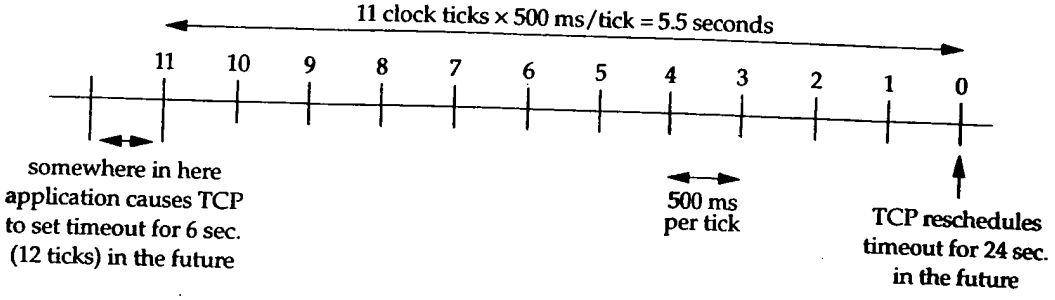


Figure 18.7 TCP 500-ms timer.

Although the timer is initialized to 12 ticks, the first decrement of the timer can occur between 0 and 500 ms after it is set. From that point on the timer is decremented about every 500 ms, but the first period can be variable. (We use the qualifier "about" because the time when TCP gets control every 500 ms can be preempted by other interrupts being handled by the kernel.)

When that 6-second timer expires at the tick labeled 0 in Figure 18.7, the timer is reset for 24 seconds (48 ticks) in the future. This next timer will be close to 24 seconds, since it was set at a time when the TCP's 500-ms timer handler was called by the kernel.

Type-of-Service Field

In Figure 18.6, the notation `[tos 0x10]` appears. This is the type-of-service (TOS) field in the IP datagram (Figure 3.2). The BSD/386 Telnet client sets the field for minimum delay.

18.4 Maximum Segment Size

The maximum segment size (MSS) is the largest "chunk" of data that TCP will send to the other end. When a connection is established, each end can announce its MSS. The values we've seen have all been 1024. The resulting IP datagram is normally 40 bytes larger: 20 bytes for the TCP header and 20 bytes for the IP header.

Some texts refer to this as a "negotiated" option. It is not negotiated in any way. When a connection is established, each end has the option of announcing the MSS it

expects to receive. (An MSS option can only appear in a SYN segment.) If one end does not receive an MSS option from the other end, a default of 536 bytes is assumed. (This default allows for a 20-byte IP header and a 20-byte TCP header to fit into a 576-byte IP datagram.)

In general, the larger the MSS the better, until fragmentation occurs. (This may not always be true. See Figures 24.3 and 24.4 for a counterexample.) A larger segment size allows more data to be sent in each segment, amortizing the cost of the IP and TCP headers. When TCP sends a SYN segment, either because a local application wants to initiate a connection, or when a connection request is received from another host, it can send an MSS value up to the outgoing interface's MTU, minus the size of the fixed TCP and IP headers. For an Ethernet this implies an MSS of up to 1460 bytes. Using IEEE 802.3 encapsulation (Section 2.2), the MSS could go up to 1452 bytes.

The values of 1024 that we've seen in this chapter, for connections involving BSD/386 and SVR4, are because many BSD implementations require the MSS to be a multiple of 512. Other systems, such as SunOS 4.1.3, Solaris 2.2, and AIX 3.2.2, all announce an MSS of 1460 when both ends are on a local Ethernet. Measurements in [Mogul 1993] show how an MSS of 1460 provides better performance on an Ethernet than an MSS of 1024.

If the destination IP address is "nonlocal," the MSS normally defaults to 536. While it's easy to say that a destination whose IP address has the same network ID and the same subnet ID as ours is local, and a destination whose IP address has a totally different network ID from ours is nonlocal, a destination with the same network ID but a different subnet ID could be either local or nonlocal. Most implementations provide a configuration option (Appendix E and Figure E.1) that lets the system administrator specify whether different subnets are local or nonlocal. The setting of this option determines whether the announced MSS is as large as possible (up to the outgoing interface's MTU) or the default of 536.

The MSS lets a host limit the size of datagrams that the other end sends it. When combined with the fact that a host can also limit the size of the datagrams that it sends, this lets a host avoid fragmentation when the host is connected to a network with a small MTU.

Consider our host `slip`, which has a SLIP link with an MTU of 296 to the router `bsdi`. Figure 18.8 shows these systems and the host `sun`.

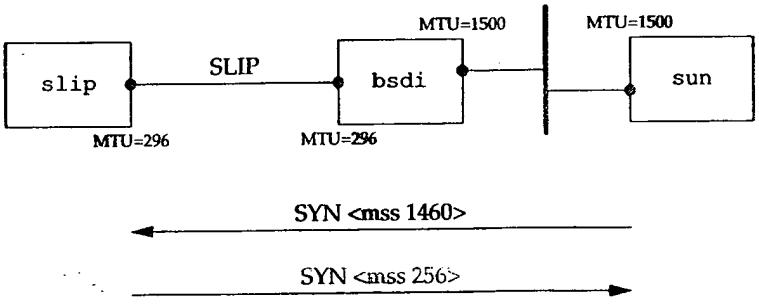


Figure 18.8 TCP connection from `sun` to `slip` showing MSS values.

We initiate a TCP connection from sun to slip and watch the segments using tcpdump. Figure 18.9 shows only the connection establishment (with the window size advertisements removed).

```
1  0.0                    sun.1093 > slip.discard: S 517312000:517312000(0)
                                                   <mss 1460>
2  0.10 (0.00)            slip.discard > sun.1093: S 509556225:509556225(0)
                                                   ack 517312001 <mss 256>
3  0.10 (0.00)            sun.1093 > slip.discard: . ack 1
```

Figure 18.9 tcpdump output for connection establishment from sun to slip.

The important fact here is that sun cannot send a segment with more than 256 bytes of data, since it received an MSS option of 256 (line 2). Furthermore, since slip knows that the outgoing interface's MTU is 296, even though sun announced an MSS of 1460, it will never send more than 256 bytes of data, to avoid fragmentation. It's OK for a system to send *less* than the MSS announced by the other end.

This avoidance of fragmentation works only if either host is directly connected to a network with an MTU of less than 576. If both hosts are connected to Ethernets, and both announce an MSS of 536, but an intermediate network has an MTU of 296, fragmentation will occur. The only way around this is to use the path MTU discovery mechanism (Section 24.2).

18.5 TCP Half-Close

TCP provides the ability for one end of a connection to terminate its output, while still receiving data from the other end. This is called a *half-close*. Few applications take advantage of this capability, as we mentioned earlier.

To use this feature the programming interface must provide a way for the application to say "I am done sending data, so send an end-of-file (FIN) to the other end, but I still want to receive data from the other end, until it sends me an end-of-file (FIN)."

> The sockets API supports the half-close, if the application calls shutdown with a second argument of 1, instead of calling close. Most applications, however, terminate both directions of the connection by calling close.

Figure 18.10 shows a typical scenario for a half-close. We show the client on the left side initiating the half-close, but either end can do this. The first two segments are the same: a FIN by the initiator, followed by an ACK of the FIN by the recipient. But it then changes from Figure 18.4, because the side that receives the half-close can still send data. We show only one data segment, followed by an ACK, but any number of data segments can be sent. (We talk more about the exchange of data segments and acknowledgments in Chapter 19.) When the end that received the half-close is done sending data, it closes its end of the connection, causing a FIN to be sent, and this delivers an end-of-file to the application that initiated the half-close. When this second FIN is acknowledged, the connection is completely closed.

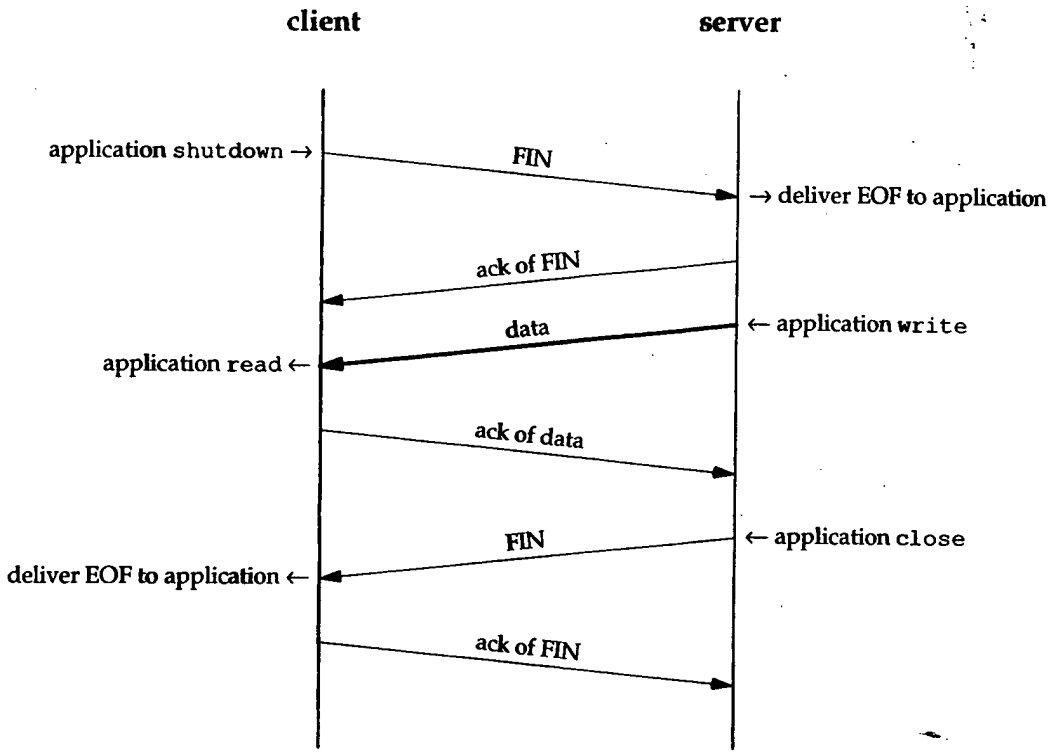


Figure 18.10 Example of TCP's half-close.

Why is there a half-close? One example is the Unix `rsh`(1) command, which executes a command on another system. The command

```
sun % rsh bsdi sort < datafile
```

executes the `sort` command on the host `bsdi` with standard input for the `rsh` command being read from the file named `datafile`. A TCP connection is created by `rsh` between itself and the program being executed on the other host. The operation of `rsh` is then simple: it copies standard input (`datafile`) to the connection, and copies from the connection to standard output (our terminal). Figure 18.11 shows the setup. (Remember that a TCP connection is full-duplex.)

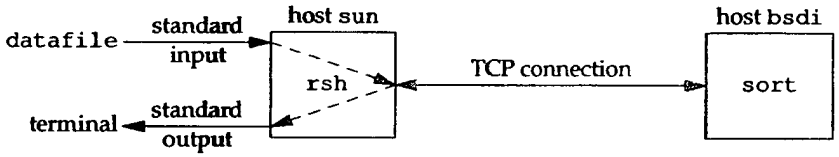


Figure 18.11 The command: `rsh bsdi sort < datafile`.

On the remote host `bsdi` the `rshd` server executes the `sort` program so that its standard input and standard output are both the TCP connection. Chapter 14 of [Stevens 1990] details the Unix process structure involved, but what concerns us here is the use of the TCP connection and the required use of TCP's half-close.

The `sort` program cannot generate any output until all of its input has been read. All the initial data across the connection is from the `rsh` client to the `sort` server, sending the file to be sorted. When the end-of-file is reached on the input (`datafile`), the `rsh` client performs a half-close on the TCP connection. The `sort` server then receives an end-of-file on its standard input (the TCP connection), sorts the file, and writes the result to its standard output (the TCP connection). The `rsh` client continues reading its end of the TCP connection, copying the sorted file to its standard output.

Without a half-close, some other technique is needed to let the client tell the server that the client is finished sending data, but still let the client receive data from the server. Two connections could be used as an alternative, but a single connection with a half-close is better.

18.6 TCP State Transition Diagram

We've described numerous rules regarding the initiation and termination of a TCP connection. These rules can be summarized in a state transition diagram, which we show in Figure 18.12.

The first thing to note in this diagram is that a subset of the state transitions is "typical." We've marked the normal client transitions with a darker solid arrow, and the normal server transitions with a darker dashed arrow.

Next, the two transitions leading to the ESTABLISHED state correspond to opening a connection, and the two transitions leading from the ESTABLISHED state are for the termination of a connection. The ESTABLISHED state is where data transfer can occur between the two ends in both directions. Later chapters describe what happens in this state.

We've collected the four boxes in the lower left of this diagram within a dashed box and labeled it "active close." Two other boxes (CLOSE_WAIT and LAST_ACK) are collected in a dashed box with the label "passive close."

The names of the 11 states (CLOSED, LISTEN, SYN_SENT, etc.) in this figure were purposely chosen to be identical to the states output by the `netstat` command. The `netstat` names, in turn, are almost identical to the names originally described in RFC 793. The state CLOSED is not really a state, but is the imaginary starting point and ending point for the diagram.

The state transition from LISTEN to SYN_SENT is legal but is not supported in Berkeley-derived implementations.

The transition from SYN_RCVD back to LISTEN is valid only if the SYN_RCVD state was entered from the LISTEN state (the normal scenario), not from the SYN_SENT state (a simultaneous open). This means if we perform a passive open (enter LISTEN), receive a SYN, send a SYN with an ACK (enter SYN_RCVD), and then receive a reset instead of an ACK, the end point returns to the LISTEN state and waits for another connection request to arrive.

Figure 18.13 shows the normal TCP connection establishment and termination, detailing the different states through which the client and server pass. It is a redo of Figure 18.3 showing only the states.

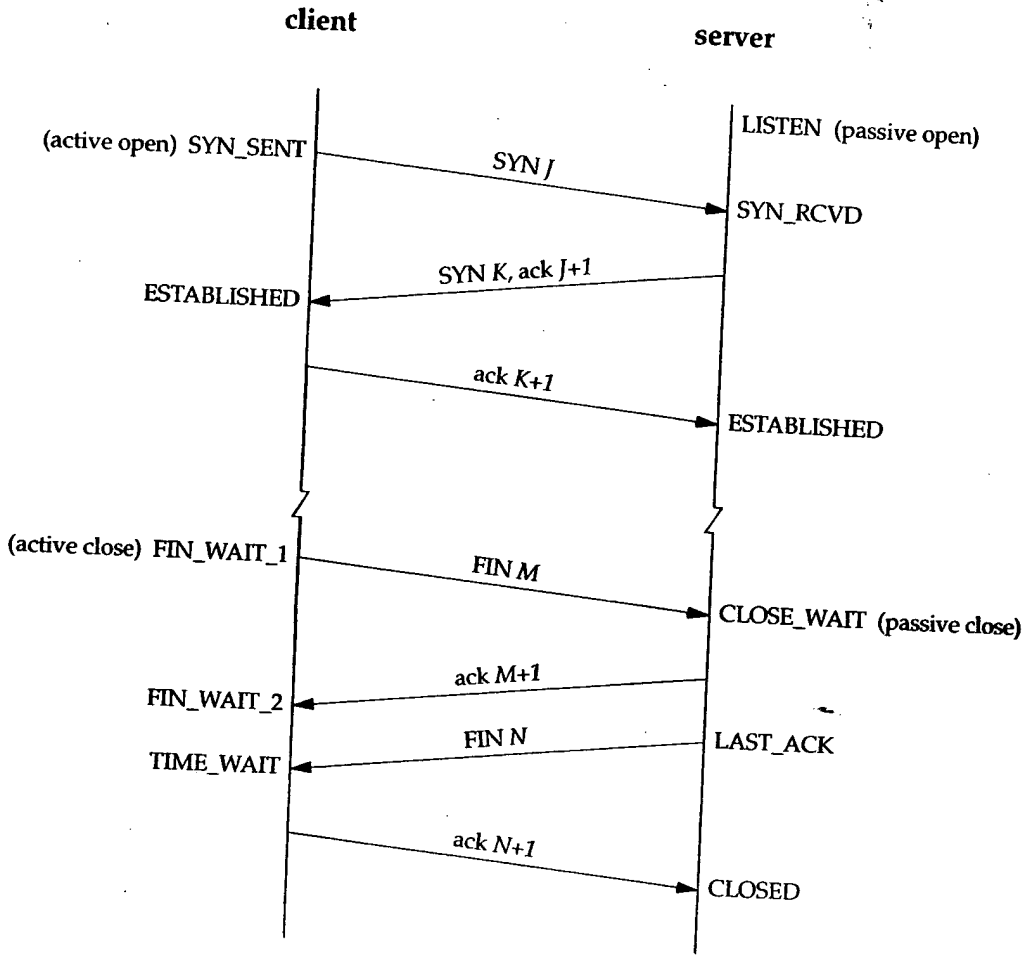


Figure 18.13 TCP states corresponding to normal connection establishment and termination.

We assume in Figure 18.13 that the client on the left side does an active open, and the server on the right side does a passive open. Although we show the client doing the active close, as we mentioned earlier, either side can do the active close.

You should follow through the state changes in Figure 18.13 using the state transition diagram in Figure 18.12, making certain you understand why each state change takes place.

2MSL Wait State

The TIME_WAIT state is also called the 2MSL wait state. Every implementation must choose a value for the *maximum segment lifetime* (MSL). It is the maximum amount of

time any segment can exist in the network before being discarded. We know this time limit is bounded, since TCP segments are transmitted as IP datagrams, and the IP datagram has the TTL field that limits its lifetime.

> RFC 793 [Postel 1981c] specifies the MSL as 2 minutes. Common implementation values, however, are 30 seconds, 1 minute, or 2 minutes.

Recall from Chapter 8 that the real-world limit on the lifetime of the IP datagram is based on the number of hops, not a timer.

Given the MSL value for an implementation, the rule is: when TCP performs an active close, and sends the final ACK, that connection must stay in the TIME_WAIT state for twice the MSL. This lets TCP resend the final ACK in case this ACK is lost (in which case the other end will time out and retransmit its final FIN).

Another effect of this 2MSL wait is that while the TCP connection is in the 2MSL wait, the socket pair defining that connection (client IP address, client port number, server IP address, and server port number) cannot be reused. That connection can only be reused when the 2MSL wait is over.

Unfortunately most implementations (i.e., the Berkeley-derived ones) impose a more stringent constraint. By default a local port number cannot be reused while that port number is the local port number of a socket pair that is in the 2MSL wait. We'll see examples of this common constraint below.

> Some implementations and APIs provide a way to bypass this restriction. With the sockets API, the `SO_REUSEADDR` socket option can be specified. It lets the caller assign itself a local port number that's in the 2MSL wait, but we'll see that the rules of TCP still prevent this port number from being part of a connection that is in the 2MSL wait.

Any delayed segments that arrive for a connection while it is in the 2MSL wait are discarded. Since the connection defined by the socket pair in the 2MSL wait cannot be reused during this time period, when we do establish a valid connection we know that delayed segments from an earlier incarnation of this connection cannot be misinterpreted as being part of the new connection. (A connection is defined by a socket pair. New instances of a connection are called *incarnations* of that connection.)

As we said with Figure 18.13, it is normally the client that does the active close and enters the TIME_WAIT state. The server usually does the passive close, and does not go through the TIME_WAIT state. The implication is that if we terminate a client, and restart the same client immediately, that new client cannot reuse the same local port number. This isn't a problem, since clients normally use ephemeral ports, and don't care what the local ephemeral port number is.

With servers, however, this changes, since servers use well-known ports. If we terminate a server that has a connection established, and immediately try to restart the server, the server cannot assign its well-known port number to its end point, since that port number is part of a connection that is in a 2MSL wait. It may take from 1 to 4 minutes before the server can be restarted.

We can see this scenario using our `sock` program. We start the server, connect to it from a client, and then terminate the server:

Quiet Time Concept

The 2MSL wait provides protection against delayed segments from an earlier incarnation of a connection from being interpreted as part of a new connection that uses the same local and foreign IP addresses and port numbers. But this works only if a host with connections in the 2MSL wait does not crash.

What if a host with ports in the 2MSL wait crashes, reboots within MSL seconds, and immediately establishes new connections using the same local and foreign IP addresses and port numbers corresponding to the local ports that were in the 2MSL wait before the crash? In this scenario, delayed segments from the connections that existed before the crash can be misinterpreted as belonging to the new connections created after the reboot. This can happen regardless of how the initial sequence number is chosen after the reboot.

To protect against this scenario, RFC 793 states that TCP should not create any connections for MSL seconds after rebooting. This is called the *quiet time*.

> Few implementations abide by this since most hosts take longer than MSL seconds to reboot after a crash.

FIN_WAIT_2 State

In the FIN_WAIT_2 state we have sent our FIN and the other end has acknowledged it. Unless we have done a half-close, we are waiting for the application on the other end to recognize that it has received an end-of-file notification and close its end of the connection, which sends us a FIN. Only when the process at the other end does this close will our end move from the FIN_WAIT_2 to the TIME_WAIT state.

This means our end of the connection can remain in this state forever. The other end is still in the CLOSE_WAIT state, and can remain there forever, until the application decides to issue its close.

> Many Berkeley-derived implementations prevent this infinite wait in the FIN_WAIT_2 state as follows. If the application that does the active close does a complete close, not a half-close indicating that it expects to receive data, then a timer is set. If the connection is idle for 10 minutes plus 75 seconds, TCP moves the connection into the CLOSED state. A comment in the code acknowledges that this implementation feature violates the protocol specification.

18.7 Reset Segments

We've mentioned a bit in the TCP header named RST for "reset." In general, a reset is sent by TCP whenever a segment arrives that doesn't appear correct for the referenced connection. (We use the term "referenced connection" to mean the connection specified by the destination IP address and port number, and the source IP address and port number. This is what RFC 793 calls a socket.)

Connection Request to Nonexistent Port

A common case for generating a reset is when a connection request arrives and no process is listening on the destination port. In the case of UDP, we saw in Section 6.5 that an ICMP port unreachable was generated when a datagram arrived for a destination port that was not in use. TCP uses a reset instead.

This example is trivial to generate—we use the Telnet client and specify a port number that's not in use on the destination:

```
bsdi % telnet svr4 20000          port 20000 should not be in use
Trying 140.252.13.34...
telnet: Unable to connect to remote host: Connection refused
```

This error message is output by the Telnet client immediately. Figure 18.14 shows the packet exchange corresponding to this command.

```
1  0.0                  bsdi.1087 > svr4.20000: S 297416193:297416193(0)
                                                 win 4096 <mss 1024>
                                                 [tos 0x10]
2  0.003771 (0.0038)    svr4.20000 > bsdi.1087: R 0:0(0) ack 297416194 win 0
```

Figure 18.14 Reset generated by attempt to open connection to nonexistent port.

The values we need to examine in this figure are the sequence number field and acknowledgment number field in the reset. Because the ACK bit was not on in the arriving segment, the sequence number of the reset is set to 0 and the acknowledgment number is set to the incoming ISN plus the number of data bytes in the segment. Although there is no real data in the arriving segment, the SYN bit logically occupies 1 byte of sequence number space; therefore, in this example the acknowledgment number in the reset is set to the ISN, plus the data length (0), plus one for the SYN bit.

Aborting a Connection

We saw in Section 18.2 that the normal way to terminate a connection is for one side to send a FIN. This is sometimes called an *orderly release* since the FIN is sent after all previously queued data has been sent, and there is normally no loss of data. But it's also possible to abort a connection by sending a reset instead of a FIN. This is sometimes called an *abortive release*.

Aborting a connection provides two features to the application: (1) any queued data is thrown away and the reset is sent immediately, and (2) the receiver of the RST can tell that the other end did an abort instead of a normal close. The API being used by the application must provide a way to generate the abort instead of a normal close.

We can watch this abort sequence happen using our `sock` program. The sockets API provides this capability by using the "linger on close" socket option (`SO_LINGER`). We specify the `-L` option with a linger time of 0. This causes the abort to be sent when the connection is closed, instead of the normal FIN. We'll connect to a server version of our `sock` program on `svr4` and type one line of input:

18.10 TCP Options

The TCP header can contain options (Figure 17.2). The only options defined in the original TCP specification are the end of option list, no operation, and the maximum segment size option. We have seen the MSS option in almost every SYN segment in our examples.

Newer RFCs, specifically RFC 1323 [Jacobson, Braden, and Borman 1992], define additional TCP options, most of which are found only in the latest implementations. (We describe these new options in Chapter 24.) Figure 18.20 shows the format of the current TCP options—those from RFC 793 and RFC 1323.

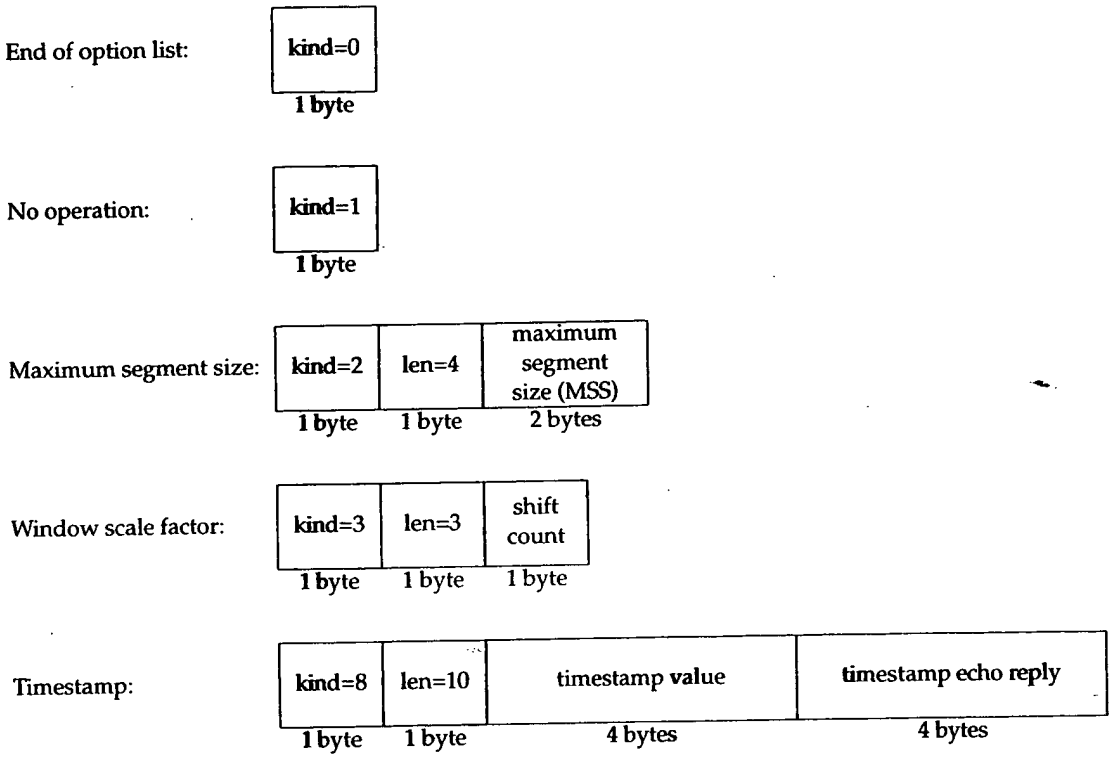


Figure 18.20 TCP options.

Every option begins with a 1-byte *kind* that specifies the type of option. The options with a *kind* of 0 and 1 occupy a single byte. The other options have a *len* byte that follows the *kind* byte. The length is the total length, including the *kind* and *len* bytes.

The reason for the no operation (NOP) option is to allow the sender to pad fields to a multiple of 4 bytes. If we initiate a TCP connection from a 4.4BSD system, the following TCP options are output by `tcpdump` on the initial SYN segment:

```
<mss 512,nop,wscale 0,nop,nop,timestamp 146647 0>
```

The MSS option is set to 512, followed by a NOP, followed by the window scale option. The reason for the first NOP is to pad the 3-byte window scale option to a 4-byte

boundary. Similarly, the 10-byte timestamp option is preceded by two NOPs, to occupy 12 bytes, placing the two 4-byte timestamps onto 4-byte boundaries.

Four other options have been proposed, with *kind*s of 4, 5, 6, and 7 called the selective-ACK and echo options. We don't show them in Figure 18.20 because the echo options have been replaced with the timestamp option, and selective ACKs, as currently defined, are still under discussion and were not included in RFC 1323. Also, the T/TCP proposal for TCP transactions (Section 24.7) specifies three options with *kind*s of 11, 12, and 13.

18.11 TCP Server Design

We said in Section 1.8 that most TCP servers are concurrent. When a new connection request arrives at a server, the server accepts the connection and invokes a new process to handle the new client. Depending on the operating system, various techniques are used to invoke the new server. Under Unix the common technique is to create a new process using the `fork` function. Lightweight processes (threads) can also be used, if supported.

What we're interested in is the interaction of TCP with concurrent servers. We need to answer the following questions: how are the port numbers handled when a server accepts a new connection request from a client, and what happens if multiple connection requests arrive at about the same time?

TCP Server Port Numbers

We can see how TCP handles the port numbers by watching any TCP server. We'll watch the Telnet server using the `netstat` command. The following output is on a system with no active Telnet connections. (We have deleted all the lines except the one showing the Telnet server.)

```
sun % netstat -a -n -f inet
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  *.23                   *.*                    LISTEN
```

The `-a` flag reports on all network end points, not just those that are ESTABLISHED. The `-n` flag prints IP addresses as dotted-decimal numbers, instead of trying to use the DNS to convert the address to a name, and prints numeric port numbers (e.g., 23) instead of service names (e.g., Telnet). The `-f inet` option reports only TCP and UDP end points.

The local address is output as `*.23`, where the asterisk is normally called the *wildcard* character. This means that an incoming connection request (i.e., a SYN) will be accepted on any local interface. If the host were multihomed, we could specify a single IP address for the local IP address (one of the host's IP addresses), and only connections received on that interface would be accepted. (We'll see an example of this later in this section.) The local port is 23, the well-known port number for Telnet.

The foreign address is output as `*.*`, which means the foreign IP address and foreign port number are not known yet, because the end point is in the LISTEN state, waiting for a connection to arrive.

We now start a Telnet client on the host `slip` (140.252.13.65) that connects to this server. Here are the relevant lines from the `netstat` output:

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.13.33.23       140.252.13.65.1029     ESTABLISHED
tcp        0      0  *.23                   *.*                    LISTEN
```

The first line for port 23 is the ESTABLISHED connection. All four elements of the local and foreign address are filled in for this connection: the local IP address and port number, and the foreign IP address and port number. The local IP address corresponds to the interface on which the connection request arrived (the Ethernet interface, 140.252.13.33).

The end point in the LISTEN state is left alone. This is the end point that the concurrent server uses to accept future connection requests. It is the TCP module in the kernel that creates the new end point in the ESTABLISHED state, when the incoming connection request arrives and is accepted. Also notice that the port number for the ESTABLISHED connection doesn't change: it's 23, the same as the LISTEN end point.

We now initiate another Telnet client from the same client (`slip`) to this server. Here is the relevant `netstat` output:

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.13.33.23       140.252.13.65.1030     ESTABLISHED
tcp        0      0  140.252.13.33.23       140.252.13.65.1029     ESTABLISHED
tcp        0      0  *.23                   *.*                    LISTEN
```

We now have two ESTABLISHED connections from the same host to the same server. Both have a local port number of 23. This is not a problem for TCP since the foreign port numbers are different. They must be different because each of the Telnet clients uses an ephemeral port, and the definition of an ephemeral port is one that is not currently in use on that host (`slip`).

This example reiterates that TCP demultiplexes incoming segments using all four values that comprise the local and foreign addresses: destination IP address, destination port number, source IP address, and source port number. TCP cannot determine which process gets an incoming segment by looking at the destination port number only. Also, the only one of the three end points at port 23 that will receive incoming connection requests is the one in the LISTEN state. The end points in the ESTABLISHED state cannot receive SYN segments, and the end point in the LISTEN state cannot receive data segments.

Next we initiate a third Telnet client, from the host `solaris` that is across the SLIP link from `sun`, and not on its Ethernet.

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.1.29.23        140.252.1.32.34603     ESTABLISHED
tcp        0      0  140.252.13.33.23       140.252.13.65.1030     ESTABLISHED
tcp        0      0  140.252.13.33.23       140.252.13.65.1029     ESTABLISHED
tcp        0      0  *.23                   *.*                    LISTEN
```

The local IP address of the first ESTABLISHED connection now corresponds to the interface address of SLIP link on the multihomed host `sun` (140.252.1.29).

20

TCP Bulk Data Flow

20.1 Introduction

In Chapter 15 we saw that TFTP uses a stop-and-wait protocol. The sender of a data block required an acknowledgment for that block before the next block was sent. In this chapter we'll see that TCP uses a different form of flow control called a *sliding window* protocol. It allows the sender to transmit multiple packets before it stops and waits for an acknowledgment. This leads to faster data transfer, since the sender doesn't have to stop and wait for an acknowledgment each time a packet is sent.

We also look at TCP's PUSH flag, something we've seen in many of the previous examples. We also look at slow start, the technique used by TCP for getting the flow of data established on a connection, and then we examine bulk data throughput.

20.2 Normal Data Flow

Let's start with a one-way transfer of 8192 bytes from the host `svr4` to the host `bsdi`. We run our `sock` program on `bsdi` as the server:

```
bsdi % sock -i -s 7777
```

The `-i` and `-s` flags tell the program to run as a "sink" server (read from the network and discard the data), and the server's port number is specified as 7777. The corresponding client is then run as:

```
svr4 % sock -i -n8 bsdi 7777
```

This causes the client to perform eight 1024-byte writes to the network. Figure 20.1 shows the time line for this exchange. We have left the first three segments in the output to show the MSS values for each end.

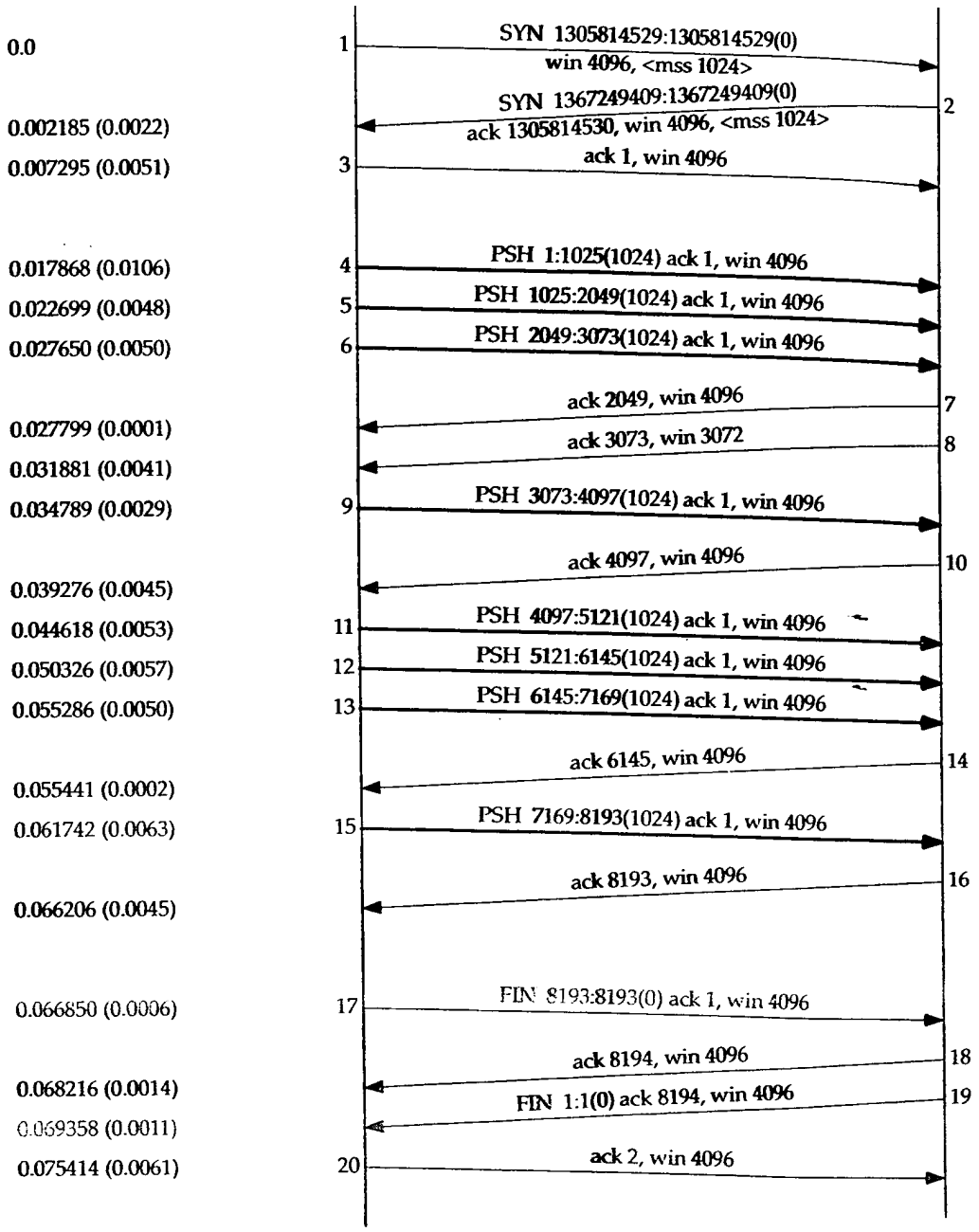


Figure 20.1 Transfer of 8192 bytes from svr4 to bsdi.

The sender transmits three data segments (4–6) first. The next segment (7) acknowledges the first two data segments only. We know this because the acknowledged sequence number is 2049, not 3073.

Segment 7 specifies an ACK of 2049 and not 3073 for the following reason. When a packet arrives it is initially processed by the device driver's interrupt service routine and then placed onto IP's input queue. The three segments 4, 5, and 6 arrive one after the other and are placed onto IP's input queue in the received order. IP will pass them to TCP in the same order. When TCP processes segment 4, the connection is marked to generate a delayed ACK. TCP processes the next segment (5) and since TCP now has two outstanding segments to ACK, the ACK of 2049 is generated (segment 7), and the delayed ACK flag for this connection is turned off. TCP processes the next input segment (6) and the connection is again marked for a delayed ACK. Before segment 9 arrives, however, it appears the delayed ACK timer goes off, and the ACK of 3073 (segment 8) is generated. Segment 8 advertises a window of 3072 bytes, implying that there are still 1024 bytes of data in the TCP receive buffer that the application has not read.

Segments 11–16 show the "ACK every other segment" strategy that is common. Segments 11, 12, and 13 arrive and are placed on IP's input queue. When segment 11 is processed by TCP the connection is marked for a delayed ACK. When segment 12 is processed, an ACK is generated (segment 14) for segments 11 and 12, and the delayed ACK flag for this connection is turned off. Segment 13 causes the connection to be marked again for a delayed ACK but before the timer goes off, segment 15 is processed, causing the ACK (segment 16) to be sent immediately.

It is important to notice that the ACK in segments 7, 14, and 16 acknowledge two received segments. With TCP's sliding-window protocol the receiver does not have to acknowledge every received packet. With TCP, the ACKs are cumulative—they acknowledge that the receiver has correctly received all bytes up through the acknowledged sequence number minus one. In this example three of the ACKs acknowledge 2048 bytes of data and two acknowledge 1024 bytes of data. (This ignores the ACKs in the connection establishment and termination.)

What we are watching with `tcpdump` are the dynamics of TCP in action. The ordering of the packets that we see on the wire depends on many factors, most of which we have no control over: the sending TCP implementation, the receiving TCP implementation, the reading of data by the receiving process (which depends on the process scheduling by the operating system), and the dynamics of the network (i.e., Ethernet collisions and backoffs). There is no single correct way for two TCPs to exchange a given amount of data.

To show how things can change, Figure 20.2 shows another time line for the same exchange of data between the same two hosts, captured a few minutes after the one in Figure 20.1.

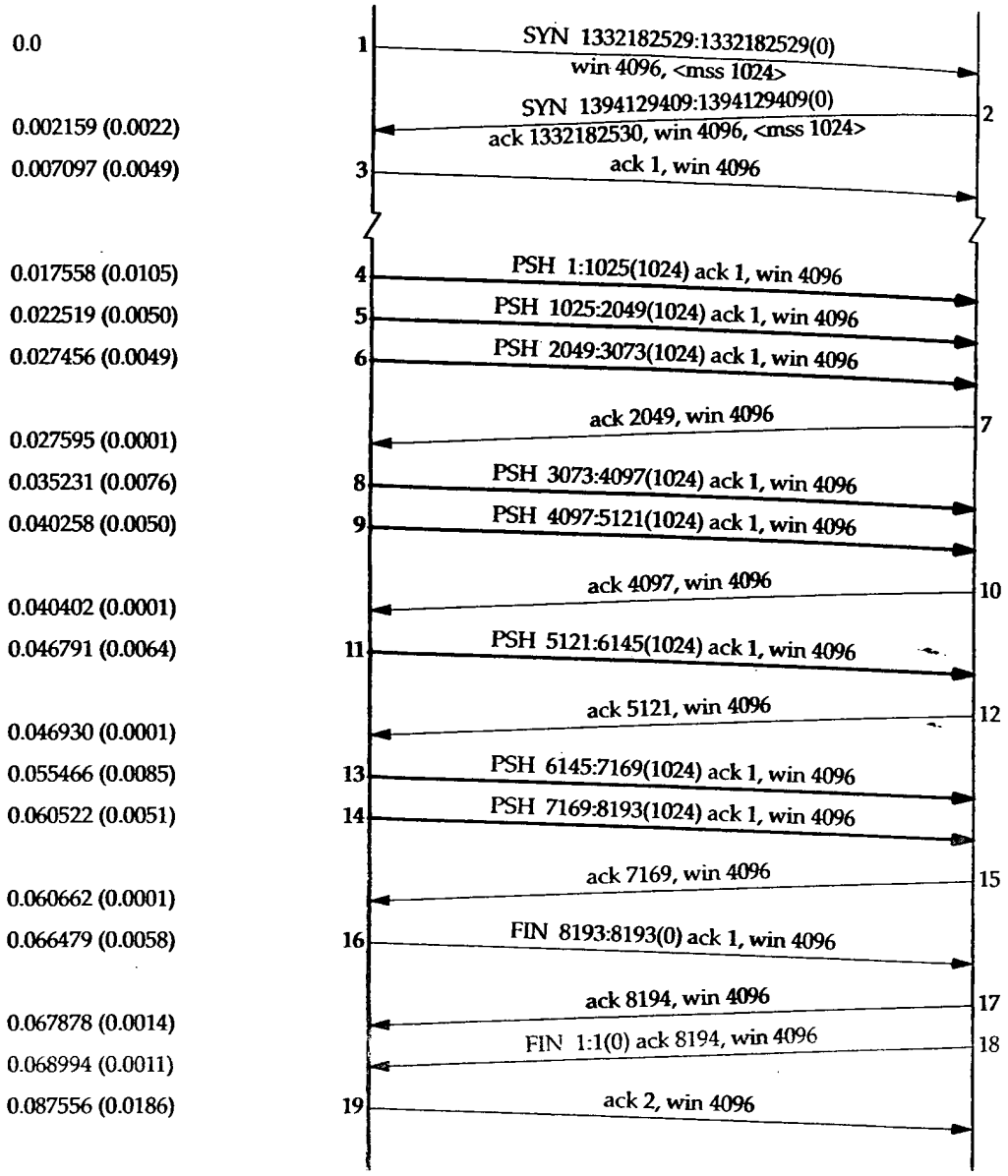


Figure 20.2 Another transfer of 8192 bytes from `svr4` to `bsdi`.

A few things have changed. This time the receiver does not send an ACK of 3073; instead it waits and sends the ACK of 4097. The receiver sends only four ACKs (segments 7, 10, 12, and 15): three of these are for 2048 bytes and one for 1024 bytes. The ACK of the final 1024 bytes of data appears in segment 17, along with the ACK of the FIN. (Compare segment 17 in this figure with segments 16 and 18 in Figure 20.1.)

Fast Sender, Slow Receiver

Figure 20.3 shows another time line, this time from a fast sender (a Sparc) to a slow receiver (an 80386 with a slow Ethernet card). The dynamics are different again.

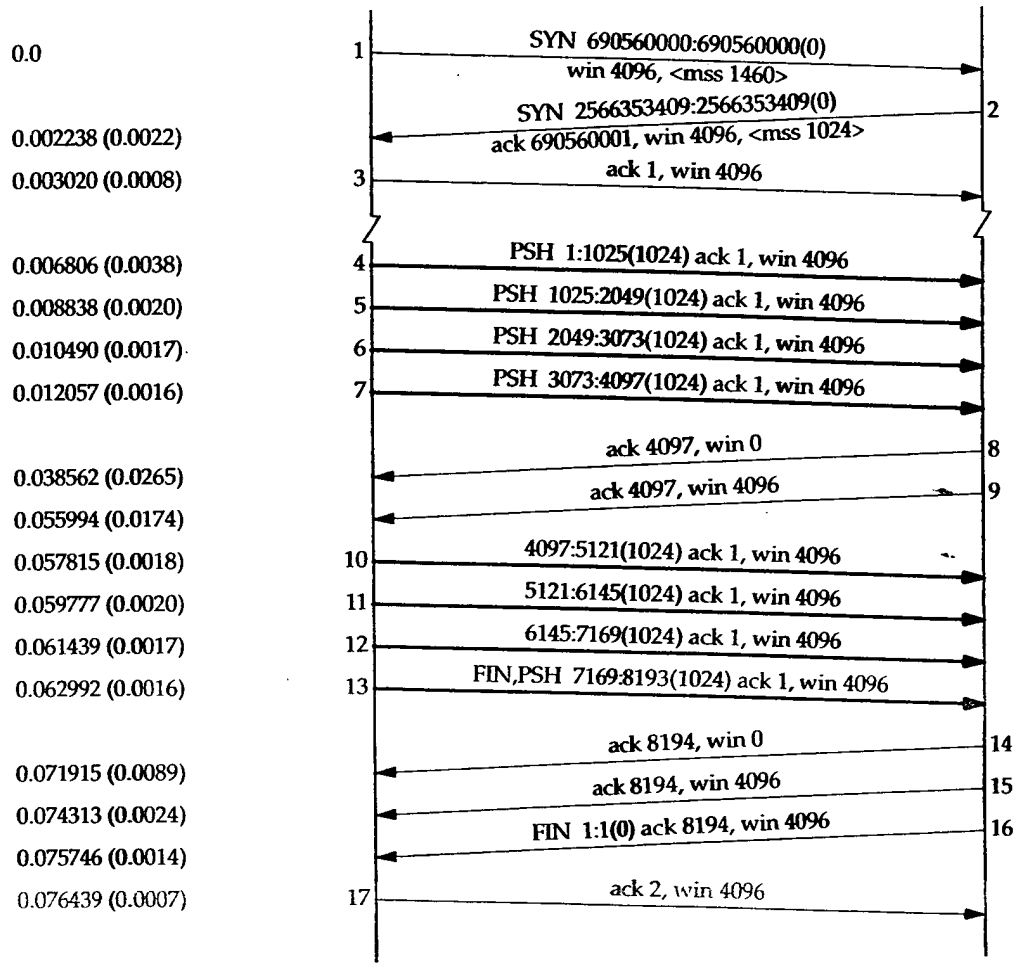


Figure 20.3 Sending 8192 bytes from a fast sender to a slow receiver.

The sender transmits four back-to-back data segments (4–7) to fill the receiver's window. The sender then stops and waits for an ACK. The receiver sends the ACK (segment 8) but the advertised window is 0. This means the receiver has all the data, but it's all in the receiver's TCP buffers, because the application hasn't had a chance to read the data. Another ACK (called a *window update*) is sent 17.4 ms later, announcing that the receiver can now receive another 4096 bytes. Although this looks like an ACK, it is called a window update because it does not acknowledge any new data, it just advances the right edge of the window.

The sender transmits its final four segments (10–13), again filling the receiver's window. Notice that segment 13 contains two flag bits: PUSH and FIN. This is followed by another two ACKs from the receiver. Both of these acknowledge the final 4096 bytes of data (bytes 4097 through 8192) and the FIN (numbered 8193).

20.3 Sliding Windows

The sliding window protocol that we observed in the previous section can be visualized as shown in Figure 20.4.

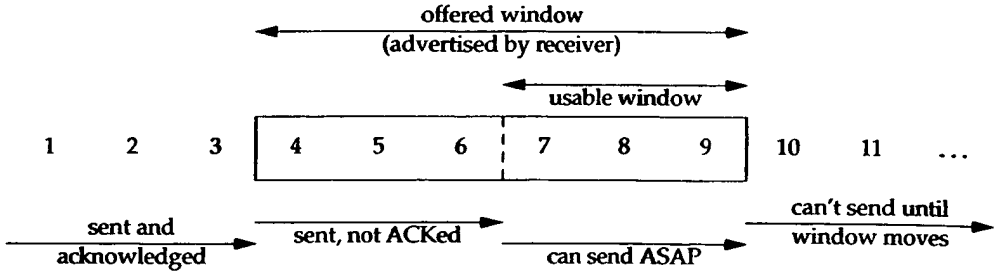


Figure 20.4 Visualization of TCP sliding window.

In this figure we have numbered the bytes 1 through 11. The window advertised by the receiver is called the *offered window* and covers bytes 4 through 9, meaning that the receiver has acknowledged all bytes up through and including number 3, and has advertised a window size of 6. Recall from Chapter 17 that the window size is relative to the acknowledged sequence number. The sender computes its *usable window*, which is how much data it can send immediately.

Over time this sliding window moves to the right, as the receiver acknowledges data. The relative motion of the two ends of the window increases or decreases the size of the window. Three terms are used to describe the movement of the right and left edges of the window.

1. The window *closes* as the left edge advances to the right. This happens when data is sent and acknowledged.
2. The window *opens* when the right edge moves to the right, allowing more data to be sent. This happens when the receiving process on the other end reads acknowledged data, freeing up space in its TCP receive buffer.
3. The window *shrinks* when the right edge moves to the left. The Host Requirements RFC strongly discourages this, but TCP must be able to cope with a peer that does this. Section 22.3 shows an example when one side would like to shrink the window by moving the right edge to the left, but cannot.

Figure 20.5 shows these three terms. The left edge of the window cannot move to the left, because this edge is controlled by the acknowledgment number received from

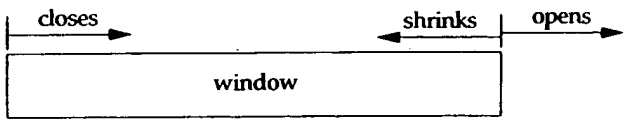


Figure 20.5 Movement of window edges.

the other end. If an ACK were received that implied moving the left edge to the left, it is a duplicate ACK, and discarded.

If the left edge reaches the right edge, it is called a *zero window*. This stops the sender from transmitting any data.

An Example

Figure 20.6 shows the dynamics of TCP's sliding window protocol for the data transfer in Figure 20.1.

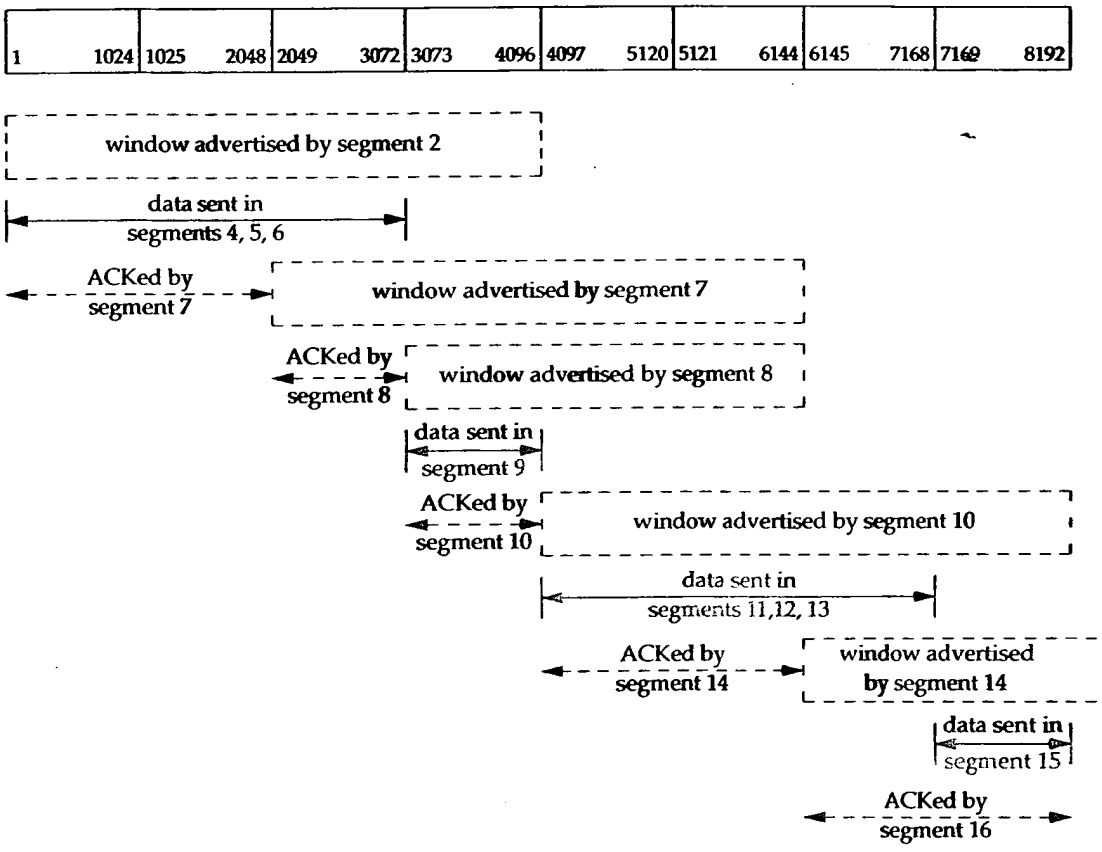


Figure 20.6 Sliding window protocol for Figure 20.1.

There are numerous points that we can summarize using this figure as an example.

1. The sender does not have to transmit a full window's worth of data.
2. One segment from the receiver acknowledges data and slides the window to the right. This is because the window size is relative to the acknowledged sequence number.
3. The size of the window can decrease, as shown by the change from segment 7 to segment 8, but the right edge of the window must not move leftward.
4. The receiver does not have to wait for the window to fill before sending an ACK. We saw earlier that many implementations send an ACK for every two segments that are received.

We'll see more examples of the dynamics of the sliding window protocol in later examples.

20.4 Window Size

The size of the window offered by the receiver can usually be controlled by the receiving process. This can affect the TCP performance.

> 4.2BSD defaulted the send buffer and receive buffer to 2048 bytes each. With 4.3BSD both were increased to 4096 bytes. As we can see from all the examples so far in this text, SunOS 4.1.3, BSD/386, and SVR4 still use this 4096-byte default. Other systems, such as Solaris 2.2, 4.4BSD, and AIX 3.2, use larger default buffer sizes, such as 8192 or 16384 bytes.
>
> The sockets API allows a process to set the sizes of the send buffer and the receive buffer. The size of the receive buffer is the maximum size of the advertised window for that connection. Some applications change the socket buffer sizes to increase performance.

[Mogul 1993] shows some results for file transfer between two workstations on an Ethernet, with varying sizes for the transmit buffer and receive buffer. (For a one-way flow of data such as file transfer, it is the size of the transmit buffer on the sending side and the size of the receive buffer on the receiving side that matters.) The common default of 4096 bytes for both is not optimal for an Ethernet. An approximate 40% increase in throughput is seen by just increasing both buffers to 16384 bytes. Similar results are shown in [Papadopoulos and Parulkar 1993].

In Section 20.7 we'll see how to calculate the minimum buffer size, given the bandwidth of the communication media and the round-trip time between the two ends.

An Example

We can control the sizes of these buffers with our `sock` program. We invoke the server as:

```
bsdi % sock -i -s -R6144 5555
```

which sets the size of the receive buffer (-R option) to 6144 bytes. We then start the client on the host sun and have it perform one write of 8192 bytes:

```
sun % sock -i -n1 -w8192 bsdi 5555
```

Figure 20.7 shows the results.

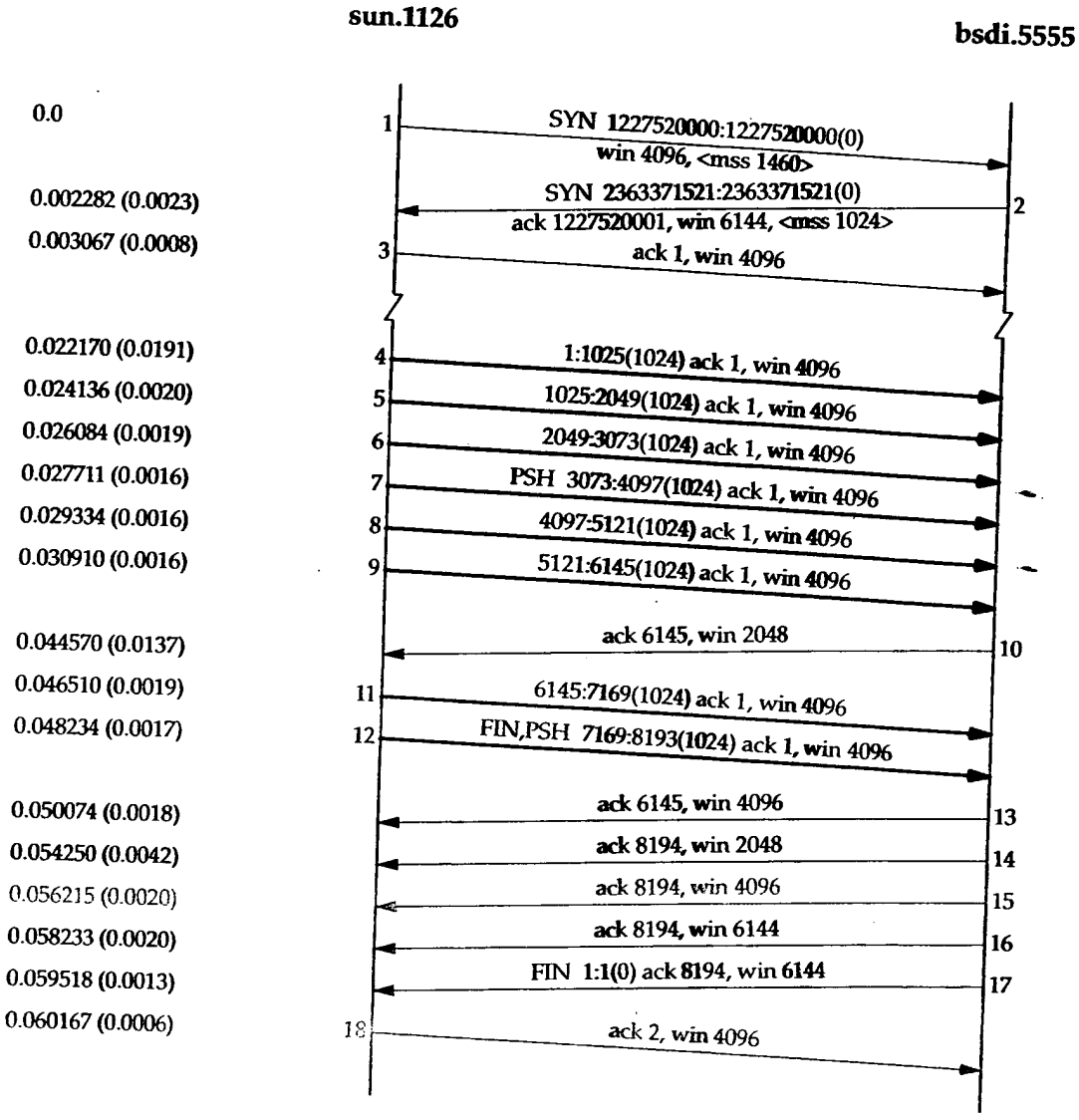


Figure 20.7 Data transfer with receiver offering a window size of 6144 bytes.

First notice that the receiver's window size is offered as 6144 bytes in segment 2. Because of this larger window, the client sends six segments immediately (segments 4–9), and then stops. Segment 10 acknowledges all the data (bytes 1 through 6144) but offers a window of only 2048, probably because the receiving application hasn't had a chance to read more than 2048 bytes. Segments 11 and 12 complete the data transfer from the client, and this final data segment also carries the FIN flag.

Segment 13 contains the same acknowledgment sequence number as segment 10, but advertises a larger window. Segment 14 acknowledges the final 2048 bytes of data and the FIN, and segments 15 and 16 just advertise a larger window. Segments 17 and 18 complete the normal close.

20.5 PUSH Flag

We've seen the PUSH flag in every one of our TCP examples, but we've never described its use. It's a notification from the sender to the receiver for the receiver to pass all the data that it has to the receiving process. This data would consist of whatever is in the segment with the PUSH flag, along with any other data the receiving TCP has collected for the receiving process.

In the original TCP specification, it was assumed that the programming interface would allow the sending process to tell its TCP when to set the PUSH flag. In an interactive application, for example, when the client sent a command to the server, the client would set the PUSH flag and wait for the server's response. (In Exercise 19.1 we could imagine the client setting the PUSH flag when the 12-byte request is written.) By allowing the client application to tell its TCP to set the flag, it was a notification to the client's TCP that the client process didn't want the data to hang around in the TCP buffer, waiting for additional data, before sending a segment to the server. Similarly, when the server's TCP received the segment with the PUSH flag, it was a notification to pass the data to the server process and not wait to see if any additional data arrives.

Today, however, most APIs don't provide a way for the application to tell its TCP to set the PUSH flag. Indeed, many implementors feel the need for the PUSH flag is outdated, and a good TCP implementation can determine when to set the flag by itself.

Most Berkeley-derived implementations automatically set the PUSH flag if the data in the segment being sent empties the send buffer. This means we normally see the PUSH flag set for each application write, because data is usually sent when it's written.

> A comment in the code indicates this algorithm is to please those implementations that only pass received data to the application when a buffer fills or a segment is received with the PUSH flag.
>
> It is not possible using the sockets API to tell TCP to turn on the PUSH flag or to tell whether the PUSH flag was set in received data.

Berkeley-derived implementations ignore a received PUSH flag because they normally never delay the delivery of received data to the application.

Examples

In Figure 20.1 (p. 276) we see the PUSH flag turned on for all eight data segments (4–6, 9, 11–13, and 15). This is because the client did eight writes of 1024 bytes, and each write emptied the send buffer.

Look again at Figure 20.7 (p. 283). We expect the PUSH flag to be set on segment 12, since that is the final data segment. Why was the PUSH flag set on segment 7, when the

sender knew there were still more bytes to send? The reason is that the size of the sender's send buffer is 4096 bytes, even though we specified a single write of 8192 bytes.

Another point to note in Figure 20.7 concerns the three consecutive ACKs, segments 14, 15, and 16. We saw two consecutive ACKs in Figure 20.3, but that was because the receiver had advertised a window of 0 (stopping the sender) so when the window opened up, another ACK was required, with the nonzero window, to restart the sender. In Figure 20.7, however, the window never reaches 0. Nevertheless, when the size of the window increases by 2048 bytes, another ACK is sent (segments 15 and 16) to provide this window update to the other end. (These two window updates in segments 15 and 16 are not needed, since the FIN has been received from the other end, meaning it will not send any more data.) Many implementations send this window update if the window increases by either two maximum sized segments (2048 bytes in this example, with an MSS of 1024) or 50% of the maximum possible window (3072 bytes in this example, with a maximum window of 6144). We'll see this again in Section 22.3 when we examine the silly window syndrome in detail.

As another example of the PUSH flag, look again at Figure 20.3 (p. 279). The reason we see the flag on for the first four data segments (4–7) is because each one caused a segment to be generated by TCP and passed to the IP layer. But then TCP had to stop, waiting for an ACK to move the 4096-byte window. While waiting for the ACK, TCP takes the final 4096 bytes of data from the application. When the window opens up (segment 9) the sending TCP knows it has four segments that it can send immediately, so it only turns on the PUSH flag for the final segment (13).

20.6 Slow Start

In all the examples we've seen so far in this chapter, the sender starts off by injecting multiple segments into the network, up to the window size advertised by the receiver. While this is OK when the two hosts are on the same LAN, if there are routers and slower links between the sender and the receiver, problems can arise. Some intermediate router must queue the packets, and it's possible for that router to run out of space. [Jacobson 1988] shows how this naive approach can reduce the throughput of a TCP connection drastically.

TCP is now required to support an algorithm called *slow start*. It operates by observing that the rate at which new packets should be injected into the network is the rate at which the acknowledgments are returned by the other end.

Slow start adds another window to the sender's TCP: the *congestion window*, called *cwnd*. When a new connection is established with a host on another network, the congestion window is initialized to one segment (i.e., the segment size announced by the other end). Each time an ACK is received, the congestion window is increased by one segment. (*cwnd* is maintained in bytes, but slow start always increments it by the segment size.) The sender can transmit up to the minimum of the congestion window and the advertised window. The congestion window is flow control imposed by the sender, while the advertised window is flow control imposed by the receiver.

The sender starts by transmitting one segment and waiting for its ACK. When that ACK is received, the congestion window is incremented from one to two, and two segments can be sent. When each of those two segments is acknowledged, the congestion window is increased to four. This provides an exponential increase.

At some point the capacity of the internet can be reached, and an intermediate router will start discarding packets. This tells the sender that its congestion window has gotten too large. When we talk about TCP's timeout and retransmission algorithms in the next chapter, we'll see how this is handled, and what happens to the congestion window. For now, let's watch slow start in action.

An Example

Figure 20.8 shows data being sent from the host `sun` to the host `vangogh.cs.berkeley.edu`. The data traverses a slow SLIP link, which should be the bottleneck. (We have removed the connection establishment from this time line.)

We see the sender transmit one segment with 512 bytes of data and then wait for its ACK. The ACK is received 716 ms later, which is an indicator of the round-trip time. The congestion window is then increased to two segments, and two segments are sent. When the ACK in segment 5 is received, the congestion window is increased to three segments. Two more segments are sent (not three) because the ACK for segment 4 is still outstanding. When the ACK in segment 8 is received, the congestion window is increased to 4 but only two more segments are sent, because the ACKs for segments 6 and 7 are still outstanding.

We'll return to slow start in Section 21.6 and see how it's normally implemented with another technique called *congestion avoidance*.

20.7 Bulk Data Throughput

Let's look at the interaction of the window size, the windowed flow control, and slow start on the throughput of a TCP connection carrying bulk data.

Figure 20.9 shows the steps over time of a connection between a sender on the left and a receiver on the right. Sixteen units of time are shown. We show only discrete units of time in this figure, for simplicity. We show segments carrying data going from the left to right in the top half of each picture, numbered 1, 2, 3, and so on. The ACKs go in the other direction in the bottom half of each picture. We draw the ACKs smaller, and show the segment number being acknowledged.

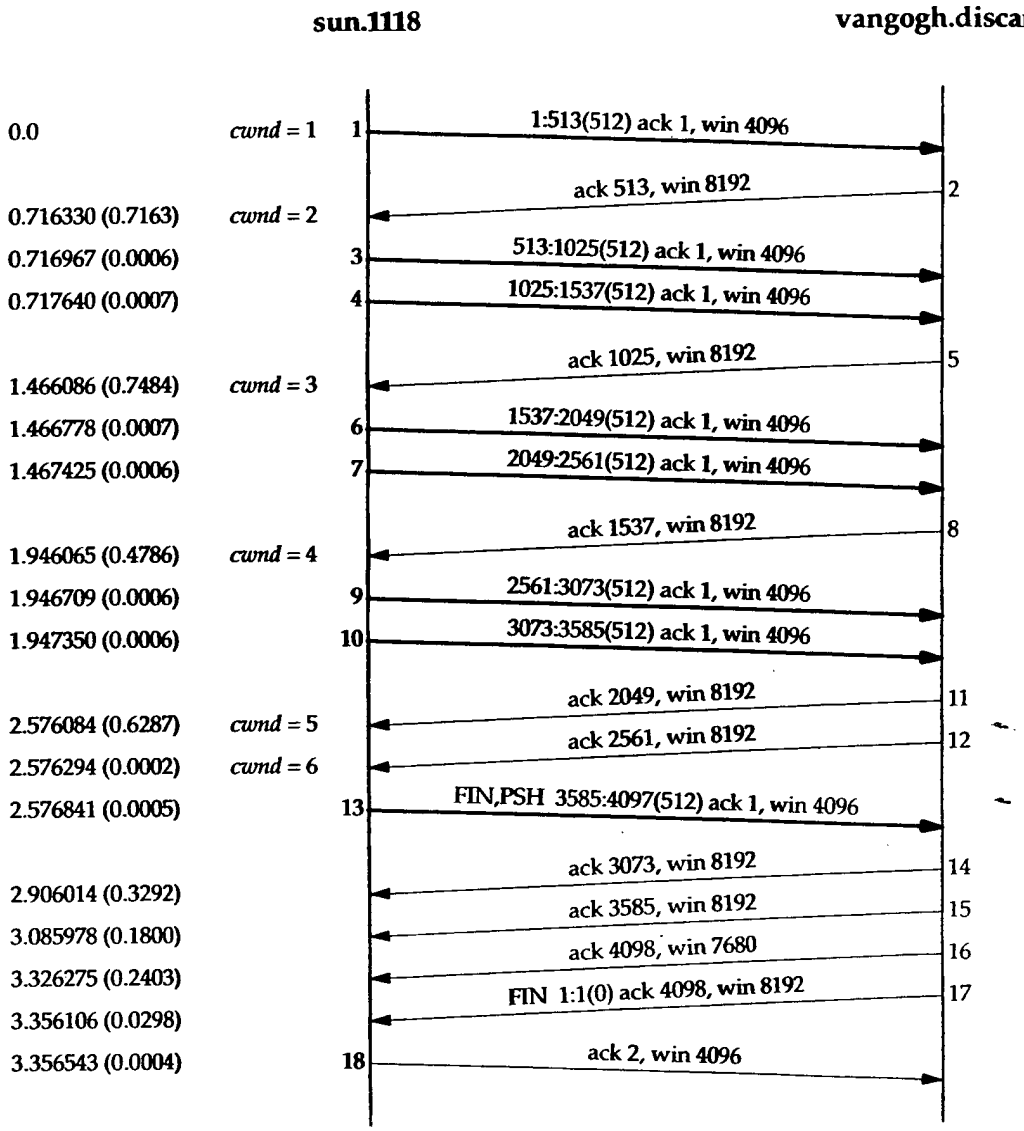


Figure 20.8 Example of slow start.

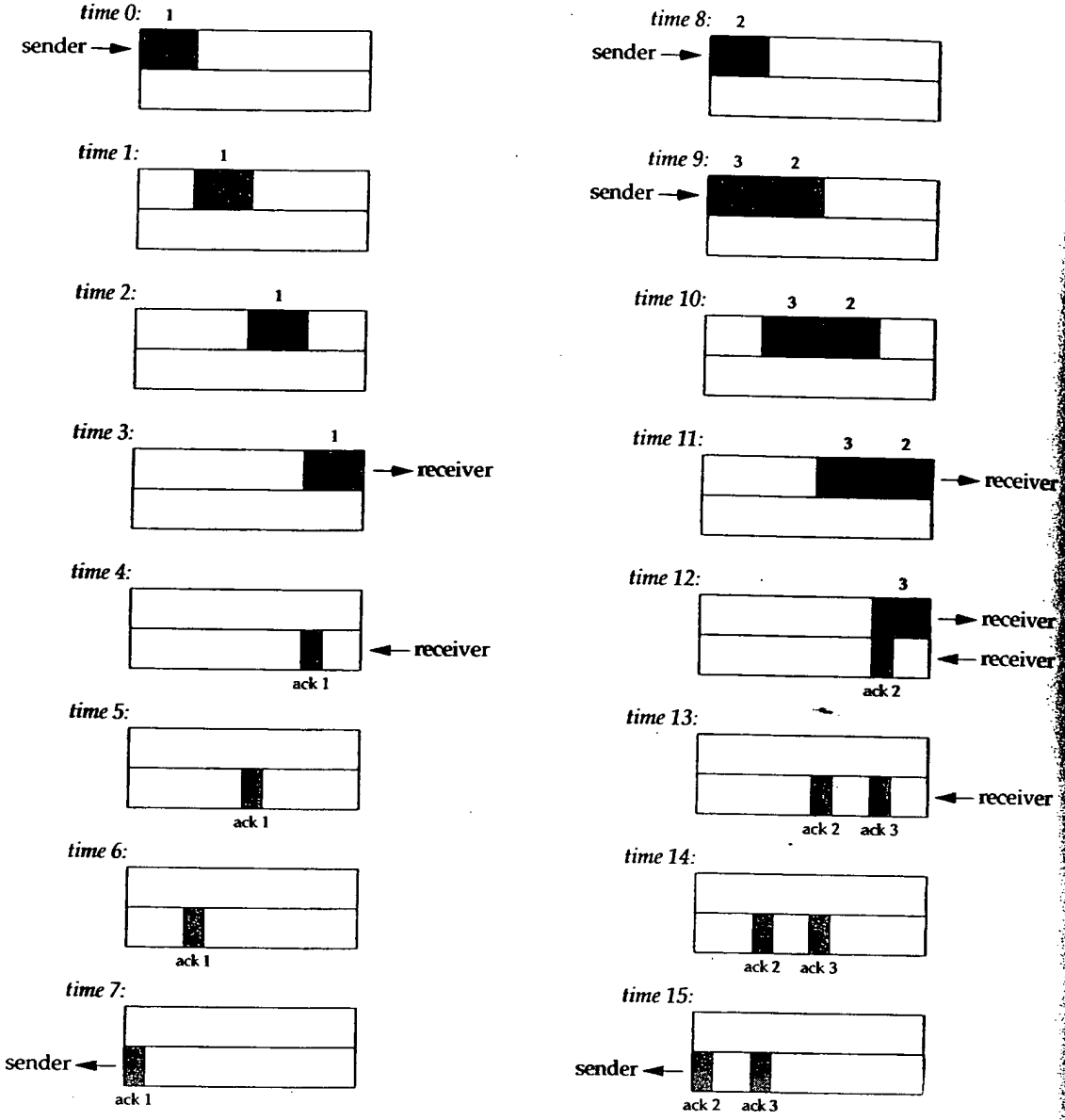


Figure 20.9 Times 0–15 for bulk data throughput example.

At time 0 the sender transmits one segment. Since the sender is in slow start (its congestion window is one segment), it must wait for the acknowledgment of this segment before continuing.

At times 1, 2, and 3 the segment moves one unit of time to the right. At time 4 the receiver reads the segment and generates the acknowledgment. At times 5, 6, and 7 the ACK moves to the left one unit, back to the sender. We have a round-trip time (RTT) of 8 units of time.

We have purposely drawn the ACK segment smaller than the data segment, since it's normally just an IP header and a TCP header. We're showing only a unidirectional

flow of data here. Also, we assume that the ACK moves at the same speed as the data segment, which isn't always true.

> In general the time to send a packet depends on two factors: a propagation delay (caused by the finite speed of light, latencies in transmission equipment, etc.) and a transmission delay that depends on the speed of the media (how many bits per second the media can transmit). For a given path between two nodes the propagation delay is fixed while the transmission delay depends on the packet size. At lower speeds the transmission delay dominates (e.g., Exercise 7.2 where we didn't even consider the propagation delay), whereas at gigabit speeds the propagation delay dominates (e.g., Figure 24.6).

When the sender receives the ACK it can transmit two more segments (which we've numbered 2 and 3), at times 8 and 9. Its congestion window is now two segments. These two segments move right toward the receiver, where the ACKs are generated at times 12 and 13. The spacing of the ACKs returned to the sender is identical to the spacing of the data segments. This is called the *self-clocking* behavior of TCP. Since the receiver can only generate ACKs when the data arrives, the spacing of the ACKs at the sender identifies the arrival rate of the data at the receiver. (In actuality, however, queueing on the return path can change the arrival rate of the ACKs.)

Figure 20.10 shows the next 16 time units. The arrival of the two ACKs increases the congestion window from two to four segments, and these four segments are sent at times 16–19. The first of the ACKs returns at time 23. The four ACKs increase the congestion window from four to eight segments, and these eight segments are transmitted at times 24–31.

At time 31, and at all successive times, the pipe between the sender and receiver is full. It cannot hold any more data, regardless of the congestion window or the window advertised by the receiver. Each unit of time a segment is removed from the network by the receiver, and another is placed into the network by the sender. However many data segments fill the pipe, there are an equal number of ACKs making the return trip. This is the ideal steady state of the connection.

Bandwidth-Delay Product

We can now answer the question: how big should the window be? In our example, the sender needs to have eight segments outstanding and unacknowledged at any time, for maximum throughput. The receiver's advertised window must be that large, since that limits how much the sender can transmit.

We can calculate the capacity of the pipe as

$$\textit{capacity}\ (\text{bits}) = \textit{bandwidth}\ (\text{bits/sec}) \times \textit{round-trip time}\ (\text{sec})$$

This is normally called the *bandwidth-delay product*. This value can vary widely, depending on the network speed and the RTT between the two ends. For example, a T1 telephone line (1,544,000 bits/sec) across the United States (about a 60-ms RTT) gives a bandwidth-delay product of 11,580 bytes. This is reasonable in terms of the buffer sizes we talked about in Section 20.4, but a T3 telephone line (45,000,000 bits/sec) across the United States gives a bandwidth-delay product of 337,500 bytes, which is bigger than the maximum allowable TCP window advertisement (65535 bytes). We describe the

24

TCP Futures and Performance

24.1 Introduction

TCP has operated for many years over data links ranging from 1200 bits/sec dialup SLIP links to Ethernets. Ethernets were the predominant form of data link for TCP/IP in the 1980s and early 1990s. Although TCP operates correctly at speeds higher than an Ethernet (T3 phone lines, FDDI, and gigabit networks, for example), certain TCP limits start to be encountered at these higher speeds.

This chapter looks at some proposed modifications to TCP that allow it to obtain the maximum throughput at these higher speeds. We first look at the path MTU discovery mechanism, which we've seen earlier in the text, focusing this time on how it operates with TCP. This often lets TCP use an MTU greater than 536 for nonlocal connections, increasing its throughput.

We then look at long fat pipes, networks that have a large bandwidth-delay product, and the TCP limits that are encountered on these networks. Two new TCP options are described that deal with long fat pipes: a window scale option (to increase TCP's maximum window above 65535 bytes) and a timestamp option. This latter option lets TCP perform more accurate RTT measurement for data segments, and also provides protection against wrapped sequence numbers, which can occur at high speeds. These two options are defined in RFC 1323 [Jacobson, Braden, and Borman 1992].

We also look at the proposed T/TCP, modifications to TCP for transactions. The transaction mode of communication features a client request responded to by a server reply. It is a common paradigm for client–server computing. The goal of T/TCP is to reduce the number of segments exchanged by the two ends, avoiding the three-way handshake and the four segments to close the connection, so that the client receives the server's reply in one RTT plus the time required to process the request.

What is impressive about these new options—path MTU discovery, the window scale option, the timestamp option, and T/TCP—is that they are backward compatible with existing TCP implementations. Newer systems that include these options can still interoperate with all older systems. With the exception of an additional field in an ICMP message that can be used by path MTU discovery, these newer options need only be implemented on the end systems that want to take advantage of them.

We finish the chapter by looking at recently published figures dealing with TCP performance.

24.2 Path MTU Discovery

In Section 2.9 we described the concept of the *path MTU*. It is the minimum MTU on any network that is currently in the path between two hosts. Path MTU discovery entails setting the "don't fragment" (DF) bit in the IP header to discover if any router on the current path needs to fragment IP datagrams that we send. In Section 11.6 we showed the ICMP unreachable error returned by a router that is asked to forward an IP datagram with the DF bit set when the MTU is less than the datagram size. In Section 11.7 we showed a version of the `traceroute` program that used this mechanism to determine the path MTU to a destination. In Section 11.8 we saw how UDP handled path MTU discovery. In this section we'll examine how this mechanism is used by TCP, as specified by RFC 1191 [Mogul and Deering 1990].

> Of the various systems used in this text (see the Preface) only Solaris 2.x supports path MTU discovery.

TCP's path MTU discovery operates as follows. When a connection is established, TCP uses the minimum of the MTU of the outgoing interface, or the MSS announced by the other end, as the starting segment size. Path MTU discovery does not allow TCP to exceed the MSS announced by the other end. If the other end does not specify an MSS, it defaults to 536. It is also possible for an implementation to save path MTU information on a per-route basis, as we mentioned in Section 21.9.

Once the initial segment size is chosen, all IP datagrams sent by TCP on that connection have the DF bit set. If an intermediate router needs to fragment a datagram that has the DF bit set, it discards the datagram and generates the ICMP "can't fragment" error we described in Section 11.6.

If this ICMP error is received, TCP decreases the segment size and retransmits. If the router generated the newer form of this ICMP error, the segment size can be set to the next-hop MTU minus the sizes of the IP and TCP headers. If the older ICMP error is returned, the probable value of the next smallest MTU (Figure 2.5) must be tried. When a retransmission caused by this ICMP error occurs, the congestion window should not change, but slow start should be initiated.

Since routes can change dynamically, when some time has passed since the last decrease of the path MTU, a larger value (up to the minimum of the MSS announced by the other end, or the outgoing interface MTU) can be tried. RFC 1191 recommends this time interval be about 10 minutes. (We saw in Section 11.8 that Solaris 2.2 uses a 30-second timer for this.)

Given the normal default MSS of 536 for nonlocal destinations, path MTU discovery avoids fragmentation across intermediate links with an MTU of less than 576 (which is rare). It can also avoid fragmentation on local destinations when an intermediate link (e.g., an Ethernet) has a smaller MTU than the end-point networks (e.g., a token ring). But for path MTU discovery to be more useful, and take advantage of wide area networks with MTUs greater than 576, implementations must stop using a default MSS of 536 bytes for nonlocal destinations. A better choice for the MSS is the MTU of the outgoing interface (minus the size of the IP and TCP headers, of course). (In Appendix E we'll see that most implementations allow the system administrator to change this default MSS value.)

An Example

We can see how path MTU discovery operates when an intermediate router has an MTU less than either of the end point's interface MTUs. Figure 24.1 shows the topology for this example.

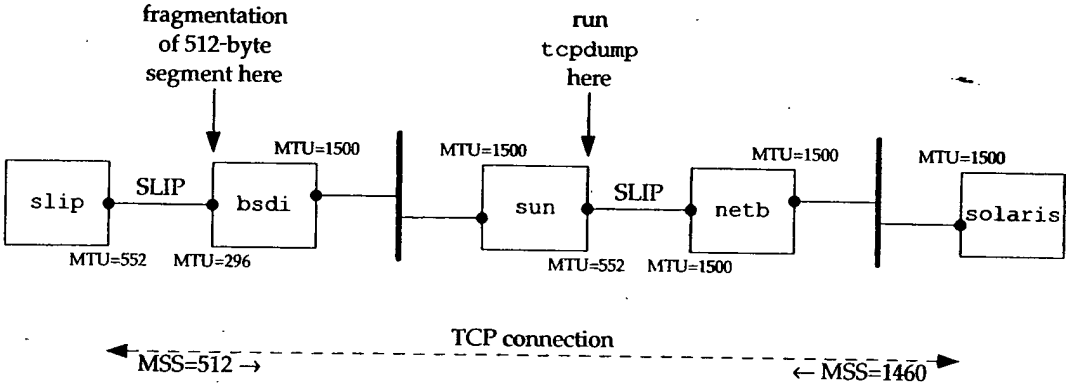


Figure 24.1 Topology for path MTU example.

We'll establish a connection from the host `solaris` (which supports the path MTU discovery mechanism) to the host `slip`. This setup is identical to the one used for our UDP path MTU discovery example (Figure 11.13) but here we have set the MTU of the interface on `slip` to 552, instead of its normal 296. This causes `slip` to announce an MSS of 512. But leaving the MTU of the SLIP link on `bsdi` at 296 will cause TCP segments greater than 256 to be fragmented, and we can see how the path MTU discovery mechanism on `solaris` handles this.

We'll run our `sock` program on `solaris` and perform one 512-byte write to the discard server on `slip`:

```
solaris % sock -i -n1 -w512 slip discard
```

Figure 24.2 shows the `tcpdump` output, collected on the SLIP interface on the host `sun`.

The MSS values in lines 1 and 2 are what we expect. We then see `solaris` send a 512-byte segment (line 3) containing the 512 bytes of data and the ACK of the SYN. (We saw this combination of the ACK of a SYN along with the first segment of data in

```
 1  0.0                  solaris.33016 > slip.discard: S 1171660288:1171660288(0)
                                                       win 8760 <mss 1460> (DF)
 2  0.101597 (0.1016)    slip.discard > solaris.33016: S 137984001:137984001(0)
                                                       ack 1171660289 win 4096
                                                       <mss 512>
 3  0.630609 (0.5290)    solaris.33016 > slip.discard: P 1:513(512)
                                                       ack 1 win 9216 (DF)
 4  0.634433 (0.0038)    bsdi > solaris: icmp:
                                 slip unreachable - need to frag, mtu = 296 (DF)
 5  0.660331 (0.0259)    solaris.33016 > slip.discard: F 513:513(0)
                                                       ack 1 win 9216 (DF)
 6  0.752664 (0.0923)    slip.discard > solaris.33016: . ack 1 win 4096
 7  1.110342 (0.3577)    solaris.33016 > slip.discard: P 1:257(256)
                                                       ack 1 win 9216 (DF)
 8  1.439330 (0.3290)    slip.discard > solaris.33016: . ack 257 win 3840
 9  1.770154 (0.3308)    solaris.33016 > slip.discard: FP 257:513(256)
                                                        ack 1 win 9216 (DF)
10  2.095987 (0.3258)    slip.discard > solaris.33016: . ack 514 win 3840
11  2.138193 (0.0422)    slip.discard > solaris.33016: F 1:1(0) ack 514 win 4096
12  2.310103 (0.1719)    solaris.33016 > slip.discard: . ack 2 win 9216 (DF)
```

Figure 24.2 `tcpdump` output for path MTU discovery.

Exercise 18.9.) This generates the ICMP error in line 4 and we see that the router `bsdi` generates the newer ICMP error containing the MTU of the outgoing interface.

It appears that before this error makes it back to `solaris`, the FIN is sent (line 5). Since `slip` never received the 512 bytes of data discarded by the router `bsdi`, it is not expecting this sequence number (513), so it responds in line 6 with the expected sequence number (1).

At this time the ICMP error has made it back to `solaris` and it retransmits the 512 bytes of data in two 256-byte segments (lines 7 and 9). Both are sent with the DF bit set, since there could be another router beyond `bsdi` with a smaller MTU.

A longer transfer was run (taking about 15 minutes) and after moving from the 512-byte initial segment to 256-byte segments, `solaris` never tried the higher segment size again.

Big Packets or Small Packets?

Conventional wisdom says that bigger packets are better [Mogul 1993, Sec. 15.2.8] because sending fewer big packets "costs less" than sending more smaller packets. (This assumes the packets are not large enough to cause fragmentation, since that introduces another set of problems.) The reduced cost is that associated with the network (packet header overhead), routers (routing decisions), and hosts (protocol processing and device interrupts). Not everyone agrees with this [Bellovin 1993].

Consider the following example. We send 8192 bytes through four routers, each connected with a T1 telephone line (1,544,000 bits/sec). First we use two 4096-byte packets, as shown in Figure 24.3.

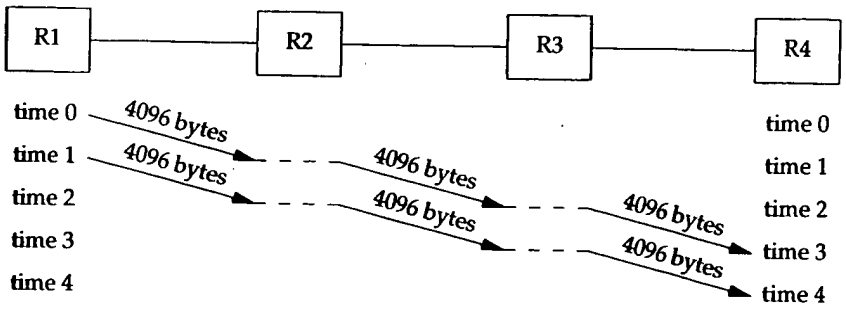


Figure 24.3 Sending two 4096-byte packets through four routers.

The basic problem is that routers are store-and-forward devices. They normally receive the entire input packet, validate the IP header including the IP checksum, make their routing decision, and start sending the output packet. In this figure we're assuming the ideal case where it takes no time for these operations to occur at the router (the horizontal dashed lines). Nevertheless, it takes four units of time to send all 8192 bytes from R1 to R4. The time for each hop is

$$\frac{(4096 + 40 \text{ bytes}) \times 8 \text{ bits/byte}}{1,544,000 \text{ bits/sec}} = 21.4 \text{ ms per hop}$$

(We account for the 40 bytes of IP and TCP header.) The total time to send the data is the number of packets plus the number of hops, minus one, which we can see visually in this example is four units of time, or 85.6 ms. Each link is idle for two units of time, or 42.8 ms.

Figure 24.4 shows what happens if we send sixteen 512-byte packets.

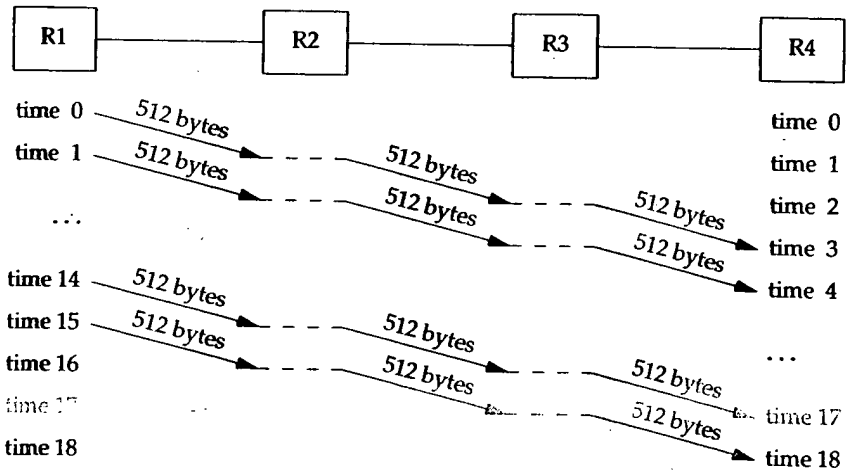


Figure 24.4 Sending sixteen 512-byte packets through four routers.

It takes more units of time, but the units are shorter since a smaller packet is being sent.

$$\frac{(512+40 \text{ bytes}) \times 8 \text{ bits/byte}}{1,544,000 \text{ bits/sec}} = 2.9 \text{ ms per hop}$$

The total time is now $(18 \times 2.9) = 52.2$ ms. Each link is again idle for two units of time, which is now 5.8 ms.

In this example we have ignored the time required for the ACKs to be returned, the connection establishment and termination times, and the possible sharing of the links with other traffic. Nevertheless, measurements in [Bellovin 1993] indicate that bigger is not always better. More research is required in this area on various networks.

24.3 Long Fat Pipes

In Section 20.7 we showed the capacity of a connection as

capacity (bits) = *bandwidth* (bits/sec) × *round-trip time* (sec)

and called this the *bandwidth-delay product*. This is also called the size of the pipe between the end points.

Existing limits in TCP are being encountered as this product increases to larger and larger values. Figure 24.5 shows some values for various types of networks.

Network	Bandwidth (bits/sec)	Round-trip time (ms)	Bandwidth-delay product (bytes)
Ethernet LAN	10,000,000	3	3,750
T1 telephone line, transcontinental	1,544,000	60	11,580
T1 telephone line, satellite	1,544,000	500	96,500
T3 telephone line, transcontinental	45,000,000	60	337,500
gigabit, transcontinental	1,000,000,000	60	7,500,000

Figure 24.5 Bandwidth-delay product for various networks.

We show the bandwidth-delay product in bytes, because that's how we typically measure the buffer sizes and window sizes required on each end.

Networks with large bandwidth-delay products are called *long fat networks* (LFNs, pronounced "elefan(t)s"), and a TCP connection operating on an LFN is called a *long fat pipe*. Going back to Figure 20.11 and Figure 20.12 (p. 291), the pipe can be stretched in the horizontal direction (a longer RTT), or stretched in the vertical direction (a higher bandwidth), or both. Numerous problems are encountered with long fat pipes.

1. The TCP window size is a 16-bit field in the TCP header, limiting the window to 65535 bytes. As we can see from the final column in Figure 24.5, existing networks already require a larger window than this, for maximum throughput.

   The window scale option described in Section 24.4 solves this problem.

2. Packet loss in an LFN can reduce throughput drastically. If only a single segment is lost, the fast retransmit and fast recovery algorithm that we described in

Section 21.7 is required to keep the pipe from draining. But even with this algorithm, the loss of more than one packet within a window typically causes the pipeline to drain. (If the pipe drains, slow start gets things going again, but that takes multiple round-trip times to get the pipe filled again.)

Selective acknowledgments (SACKs) were proposed in RFC 1072 [Jacobson and Braden 1988] to handle multiple dropped packets within a window. But this feature was omitted from RFC 1323, because the authors felt several technical problems needed to be worked out before including them in TCP.

3. We saw in Section 21.4 that many TCP implementations only measure one round-trip time per window. They do not measure the RTT of every segment. Better RTT measurements are required for operating on an LFN.

   The timestamp option, which we describe in Section 24.5, allows more segments to be timed, including retransmissions.

4. TCP identifies each byte of data with a 32-bit unsigned sequence number. What's to prevent a segment that gets delayed in the network from reappearing at a later time, after the connection that it was associated with has been terminated, and after a new connection has been established between the same two hosts and port numbers? First recall that the TTL field in the IP header puts an upper bound on the lifetime of any IP datagram—255 hops or 255 seconds, whichever comes first. In Section 18.6 we defined the maximum segment lifetime (MSL) as an implementation parameter used to prevent this scenario from happening. The recommended value of the MSL is 2 minutes (giving a 2MSL of 240 seconds), but we saw in Section 18.6 that many implementations use an MSL value of 30 seconds.

   A different problem with TCP's sequence numbers appears with LFNs. Since the sequence number space is finite, the same sequence number is reused after 4,294,967,296 bytes have been transmitted. What if a segment containing the byte with a sequence number N gets delayed in the network and then reappears later, while the connection is still up? This is only a problem if the same sequence number N is reused within the MSL period, that is, if the network is so fast that sequence number wrap occurs in less than MSL. On an Ethernet it takes almost 60 minutes to send this much data, so there is no chance of this happening, but the time required for the wrap to occur drops as the bandwidth increases: a T3 telephone line (45 Mbits/sec) wraps in 12 minutes, FDDI (100 Mbits/sec) in 5 minutes, and a gigabit network (1000 Mbits/sec) in 34 seconds. The problem here is not the bandwidth-delay product, but the bandwidth itself.

   In Section 24.6 we describe a way to handle this: the PAWS algorithm (protection against wrapped sequence numbers), which uses the TCP timestamp option.

4.4BSD contains all the options and algorithms that we describe in the following sections: the window scale option, the timestamp option, and the protection against wrapped sequence numbers. Numerous vendors are also starting to support these options.

Gigabit Networks

When networks reach gigabit speeds, things change. [Partridge 1994] covers gigabit networks in detail. Here we'll look at the differences between latency and bandwidth [Kleinrock 1992].

Consider sending a one million byte file across the United States, assuming a 30-ms latency. Figure 24.6 shows two scenarios, the top illustration uses a T1 telephone line (1,544,000 bits/sec) and the bottom uses a 1 gigabit/sec network. Time is shown along the x-axis, with the sender on the left and the receiver on the right, and capacity on the y-axis. The shaded area in both pictures is the one million bytes to send.

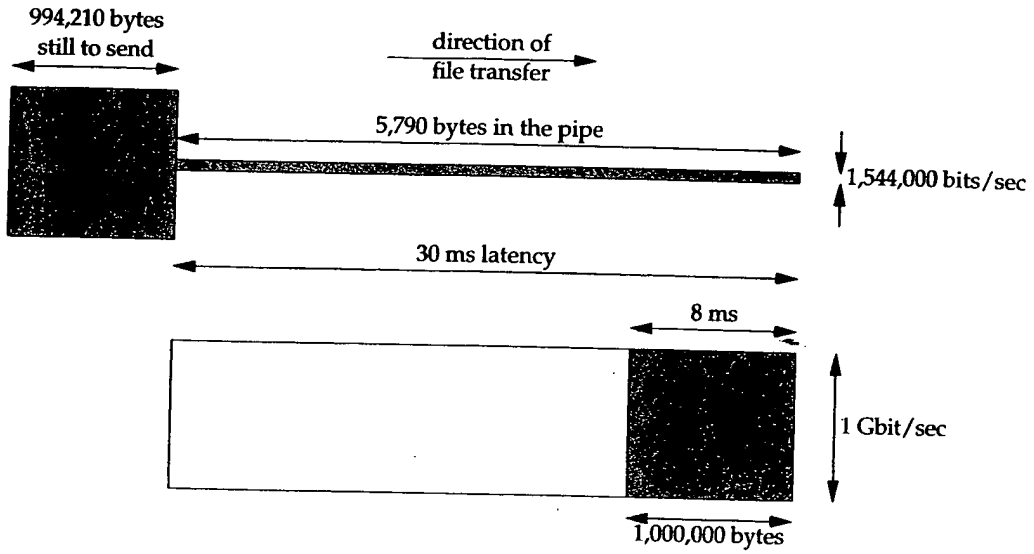


Figure 24.6 Sending a 1-Mbyte file across networks with a 30-ms latency.

Figure 24.6 shows the status of both networks after 30 ms. With both networks the first bit of data reaches the other end after 30 ms (the latency), but with the T1 network the capacity of the pipe is only 5,790 bytes, so 994,210 bytes are still at the sender, waiting to be sent. The capacity of the gigabit network, however, is 3,750,000 bytes, so the entire file uses just over 25% of the pipe. The last bit of the file reaches the receiver 8 ms after the first bit.

The total time to transfer the file across the T1 network is 5.211 seconds. If we throw more bandwidth at the problem, a T3 network (45,000,000 bits/sec), the total time decreases to 0.208 seconds. Increasing the bandwidth by a factor of 29 reduces the total time by a factor of 25.

With the gigabit network the total time to transfer the file is 0.038 seconds: the 30-ms latency plus the 8 ms for the actual file transfer. Assuming we could double the bandwidth to 2 gigabits/sec, we only reduce the total time to 0.034 seconds: the same 30-ms latency plus 4 ms to transfer the file. Doubling the bandwidth now decreases the total time by only 10%. At gigabit speeds we are latency limited, not bandwidth limited.

The latency is caused by the speed of light and can't be decreased (unless Einstein was wrong). The effect of this fixed latency becomes worse when we consider the packets required to establish and terminate a connection. Gigabit networks will cause several networking issues to be looked at differently.

24.4 Window Scale Option

The window scale option increases the definition of the TCP window from 16 to 32 bits. Instead of changing the TCP header to accommodate the larger window, the header still holds a 16-bit value, and an option is defined that applies a scaling operation to the 16-bit value. TCP then maintains the "real" window size internally as a 32-bit value.

We saw an example of this option in Figure 18.20 (p. 253). The 1-byte shift count is between 0 (no scaling performed) and 14. This maximum value of 14 is a window of 1,073,725,440 bytes (65535×2^{14}).

This option can only appear in a SYN segment; therefore the scale factor is fixed in each direction when the connection is established. To enable window scaling, both ends must send the option in their SYN segments. The end doing the active open sends the option in its SYN, but the end doing the passive open can send the option only if the received SYN specifies the option. The scale factor can be different in each direction.

If the end doing the active open sends a nonzero scale factor, but doesn't receive a window scale option from the other end, it sets its send and receive shift count to 0. This lets newer systems interoperate with older systems that don't understand the new option.

> The Host Requirements RFC requires TCP to accept an option in any segment. (The only previously defined option, the maximum segment size, only appeared in SYN segments.) It further requires TCP to ignore any option it doesn't understand. This is made easy since all the new options have a length field (Figure 18.20, p. 253).

Assume we are using the window scale option, with a shift count of S for sending and a shift count of R for receiving. Then every 16-bit advertised window that we receive from the other end is left shifted by R bits to obtain the real advertised window size. Every time we send a window advertisement to the other end, we take our real 32-bit window size and right shift it S bits, placing the resulting 16-bit value in the TCP header.

The shift count is automatically chosen by TCP, based on the size of the receive buffer. The size of this buffer is set by the system, but the capability is normally provided for the application to change it. (We discussed this buffer in Section 20.4.)

An Example

If we initiate a connection using our `sock` program from the 4.4BSD host `vangogh.cs.berkeley.edu`, we can see its TCP calculate the window scale factor. The following interactive output shows two consecutive runs of the program, first specifying a receive buffer of 128000 bytes, and then a receive buffer of 220000 bytes:

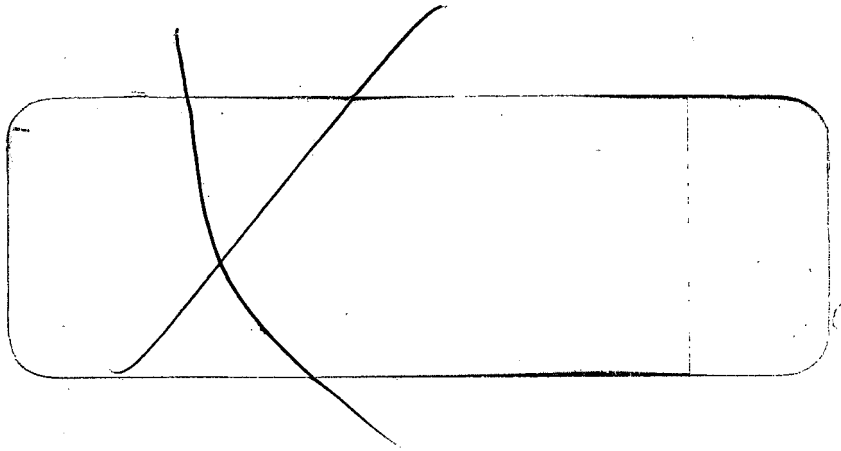
Organization 1C2100 Bldg./Room RANDOLPH

U. S. DEPARTMENT OF COMMERCE
COMMISSIONER FOR PATENTS
P.O. BOX 1450
ALEXANDRIA, VA 22313-1450
IF UNDELIVERABLE RETURN IN TEN DAYS

OFFICIAL BUSINESS

AN EQUAL

U.S. OFFICIAL MAIL
PENALTY FOR PRIVATE USE $300
$ 04.20[illegible]
MAILED FROM ZIPCODE 22314



RETURNED TO SENDER
UNDELIVERABLE AS ADDRESSED
UNABLE TO FORWARD

RECEIVED
FEB 0 6 2006
USPTO MAIL CENTER